9

METAPROGRAMMING

One of the most confusing and daunting subjects Ruby developers face is *metaprogramming*. Metaprogramming, as indicated by the prefix *meta*, literally means to program at a different or higher level of abstraction. Ruby provides many different ways for you to do this, allowing your program to inspect and change itself dynamically. In Ruby, your program can change itself!

Some of Ruby's metaprogramming features allow your program to query for information about itself—for example, information about methods, instance variables, and superclasses. Other metaprogramming features allow you to perform normal tasks, such as defining a method or a constant, in an alternative and more flexible manner. Finally, methods such as `eval` allow your program to write new Ruby code from scratch, calling the parser and compiler at run time.

In this chapter, we'll focus on two important aspects of metaprogramming. First, we'll look at how you can alter the standard method definition process, the most common and practical use for metaprogramming. We'll learn what Ruby normally does to assign a method to a class and how this is related to lexical scope. Then, we'll look at alternative ways to define methods using metaclasses and singleton classes. We'll also learn how Ruby implements the new, experimental refinements feature, allowing you to define methods and activate them later if you wish.

ROADMAP

In the second half of this chapter, we'll see how you can write code that writes code with the `eval` method: metaprogramming in its purest form. We'll also see how metaprogramming and closures are related. Like blocks, lambdas, and procs, `eval` and its related metaprogramming methods create a closure when you call them. In fact, we'll learn how you can use the same mental model we developed in Chapter 8 for blocks to understand many of Ruby's metaprogramming features.

Alternative Ways to Define Methods

Normally we define methods in Ruby using the `def` keyword. After `def`, we specify a name for the new method followed by the method body. By using some of Ruby's metaprogramming features, however, we can define methods in alternative ways. We can create class methods instead of normal methods; we can create methods for a single object instance; and, as we'll see in Experiment 9-2, we can create methods that can access the surrounding environment using a closure.

Next, we'll look at what happens inside Ruby when you define a method in each of these ways using metaprogramming. In each case, studying what Ruby does internally will make Ruby's metaprogramming syntax easier to understand. But before we tackle metaprogramming, let's learn more about how Ruby normally defines a method. This knowledge will serve as a foundation when we learn alternative ways to define a method.

Ruby's Normal Method Definition Process

Listing 9-1 shows a very simple Ruby class containing a single method.

```
class Quote
  def display
    puts "The quick brown fox jumped over the lazy dog."
  end
end
```

Listing 9-1: Adding a method to a class using the `def` keyword

How does Ruby execute this small program? And how does it know to assign the `display` method to the `Quote` class?

When Ruby executes the `class` keyword, it creates a new lexical scope for the new `Quote` class (see Figure 9-1). Ruby sets the `nd_clss` pointer in the lexical scope to point to an `RClass` structure for the new `Quote` class. Because it's a new class, the `RClass` structure initially has an empty method table, as shown on the right side of the figure.

Next, Ruby executes the `def` keyword, which is used to define the `display` method. But how does Ruby create normal methods? What happens internally when you call `def`?

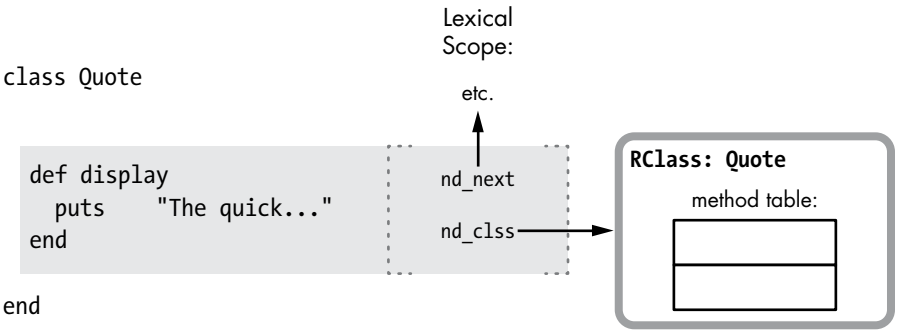

Figure 9-1: Ruby creates a new lexical scope when you define a class.

By default, when you use `def`, you provide just the name of the new method. (We'll see in the next section that you can also specify an object prefix along with the new method name.) Providing just the name of the new method with `def` instructs Ruby to use the current lexical scope to find the target class, as shown in Figure 9-2.

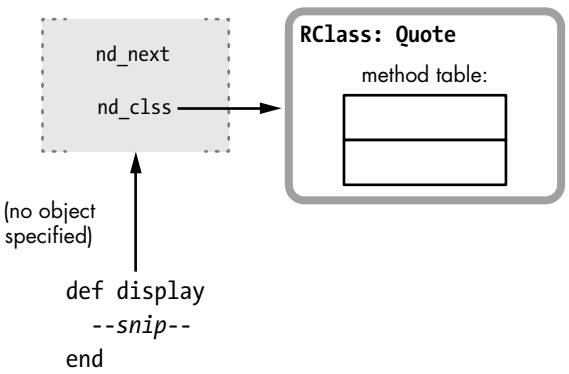

Figure 9-2: By default, Ruby uses the current lexical scope to find the target class for a new method.

When Ruby initially compiles Listing 9-1, it creates a separate snippet of YARV code for the `display` method. Later, when executing the `def` keyword, Ruby assigns this code to the target class, `Quote`, saving the given method name in the method table (see Figure 9-3).

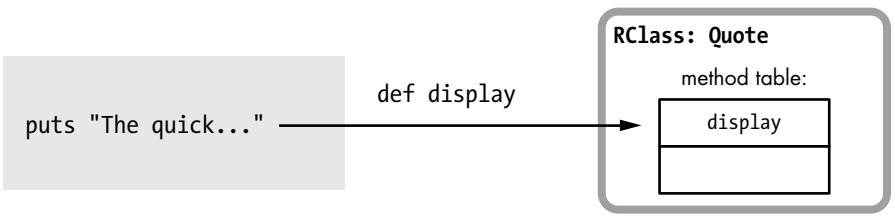

Figure 9-3: Ruby adds new methods to the method table for the target class.

When we execute this method, Ruby looks up the method as described in "Ruby's Method Lookup Algorithm" on page 138. Because `display` now appears in the method table for `Quote`, Ruby can find the method and execute it.

In sum, to define new methods in your program using the `def` keyword, Ruby follows this three-step process:

1. It compiles each method's body into a distinct snippet of YARV instructions. (This occurs when Ruby parses and compiles your program.)
2. It uses the current lexical scope to obtain a pointer to a class or module. (This occurs when Ruby encounters a `def` keyword while executing your program.)
3. It saves the new method's name—actually, an integer ID value that maps to the name—in the method table for that class.

Defining Class Methods Using an Object Prefix

Now that we understand how Ruby's method definition process normally works, let's learn alternative ways to define methods using metaprogramming. As we saw in Figure 9-2, Ruby normally assigns new methods to the class that corresponds to the current lexical scope. However, sometimes you'll decide to add a method to a different class—for example, when you define a class method. (Remember that Ruby saves class methods in a class's metaclass.) Listing 9-2 shows an example of creating a class method.

```
class Quote
❶  def self.display
     puts "The quick brown fox jumped over the lazy dog."
   end
end
```

Listing 9-2: Adding a class method using `def self`

At ❶ we use `def` to define the new method, but this time we use a `self` prefix. This prefix tells Ruby to add the method to the class of the object you specify in the prefix rather than using the current lexical scope. Figure 9-4 shows how Ruby does this internally.

This behavior is very different from the standard method definition process! When you provide an object prefix to `def`, Ruby uses the following algorithm to decide where to put the new method:

1. Ruby evaluates the prefix expression. In Listing 9-2 we use the `self` keyword. While Ruby is executing code inside the `class Quote` scope, `self` is set to the `Quote` class. (We could have provided any Ruby expression here instead of `self`.) In Figure 9-4, the arrow extending up from `self` to the `RClass` structure indicates the value of `self` is `Quote`.

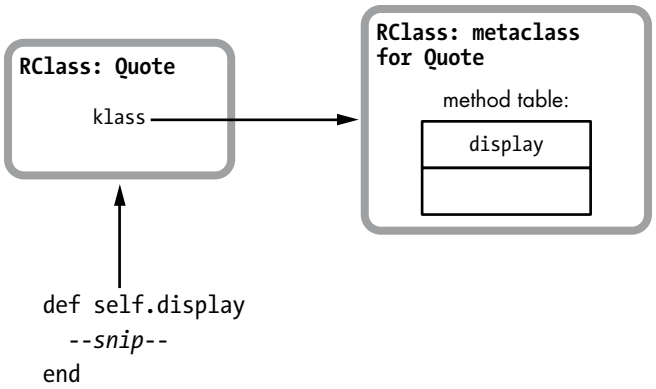

Figure 9-4: Providing an object prefix to `def` instructs Ruby to add the new method to the object's class.

2. Ruby finds the class of this object. In Listing 9-2, because `self` is a class itself (`Quote`), the class of the object is actually the metaclass for `Quote`. Figure 9-4 indicates this with the arrow extending to the right from the `RClass` structure for `Quote`.
3. Ruby saves the new method in that class's method table. In this case, Ruby places the `display` method in the metaclass for `Quote`, making `display` a new class method.

NOTE *If you call `Quote.class`, Ruby will return `Class`. All classes are officially instances of the `Class` class. Metaclasses are an internal concept, normally hidden from your Ruby program. To see the metaclass for `Quote`, you can call `Quote.singleton_class` instead, which will return `#<Class:Quote>`.*

Defining Class Methods Using a New Lexical Scope

Listing 9-3 shows a different way to assign `display` as a class method of `Quote`.

```
❶ class Quote
❷   class << self
      def display
        puts "The quick brown fox jumped over the lazy dog."
      end
    end
  end
```

Listing 9-3: Defining a class method using `class << self`

At ❷ `class << self` declares a new lexical scope, just as `class Quote` does at ❶. In "Ruby's Normal Method Definition Process" on page 221, we saw that using `def` in the scope created by `class Quote` assigns new methods to

Quote. But what class does Ruby assign methods to inside the scope created by class << self? The answer is self's class. Because at ❷ self is set to Quote, self's class is the metaclass of Quote.

Figure 9-5 shows how class << self creates a new lexical scope for the metaclass of Quote.

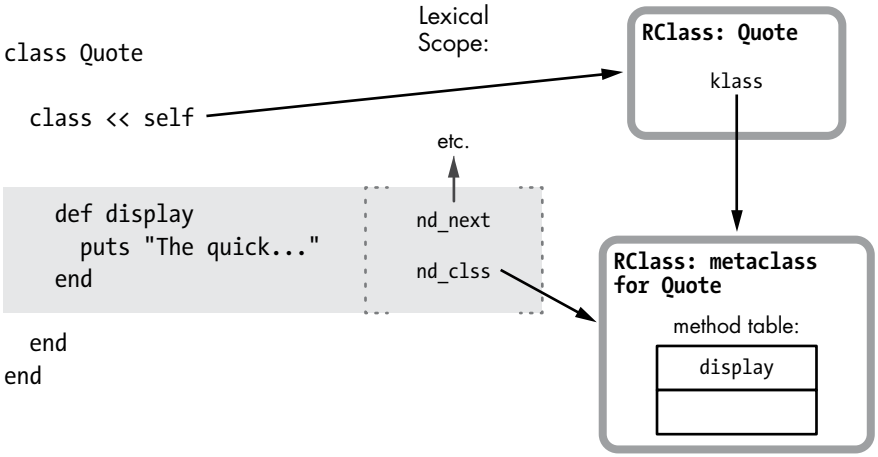

Figure 9-5: Ruby creates a new lexical scope for a class's metaclass when you use class << self.

In this figure, Ruby's class << metaprogramming syntax functions as follows:

1. Ruby first evaluates the expression that appears after class <<. In Listing 9-3 this is the expression self, which evaluates to the Quote class, just as it did using the object prefix syntax in Listing 9-2. The long arrow extending to the right from self to the RClass structure indicates the value of self is the Quote class.
2. Ruby finds the class for the object the expression evaluates to. In Listing 9-3 this will be the class of Quote, or Quote's metaclass, indicated by the arrow extending down from Quote to the metaclass for Quote on the right side of the figure.
3. Ruby creates a new lexical scope for this class. In this example, the lexical scope uses the metaclass of Quote, indicated by the arrow extending to the right from nd_clss in the new scope.

Now we can use the new lexical scope to define a series of class methods using def as usual. In Listing 9-3 Ruby will assign the display method directly to the metaclass of Quote. This is a different way of defining a class method for Quote. You might find class << self a bit more confusing than def self, but it is a convenient way to create a series of class methods by declaring them all inside the inner, metaclass lexical scope.

Defining Methods Using Singleton Classes

We've seen how metaprogramming allows you to declare class methods by adding methods to the class's class or metaclass. Ruby also allows you to add methods to a single object instance, as shown in Listing 9-4.

```
❶ class Quote
  end

❷ some_quote = Quote.new
❸ def some_quote.display
    puts "The quick brown fox jumped over the lazy dog."
  end
```

Listing 9-4: Adding a method to a single object instance

At ❶ we declare the `Quote` class; then, at ❷ we create an instance of `Quote`: `some_quote`. At ❸ this time, however, we create a new method for the `some_quote` instance, not the `Quote` class. As a result, only `some_quote` will have the `display` method; no other instances of `Quote` will have it.

Internally, Ruby implements this behavior using a hidden class called the *singleton class*, which is like a metaclass for a single object. Here's the difference:

- A *singleton class* is a special hidden class that Ruby creates internally to hold methods defined only for a particular object.
- A *metaclass* is a singleton class in the case when that object is itself a class.

All metaclasses are singleton classes, but not all singleton classes are metaclasses. Ruby automatically creates a metaclass for every class you create and uses it to hold class methods that you might declare later. On the other hand, Ruby creates a singleton class only when you define a method on a single object, as shown in Listing 9-4. Ruby also creates a singleton class when you use `instance_eval` or related methods.

NOTE *Most Ruby developers use the terms* singleton class *and* metaclass *interchangeably, and when you call the* `singleton_class` *method, Ruby will return either a singleton class or a metaclass. However, internally Ruby's C source code does make a distinction between singleton classes and metaclasses.*

Figure 9-6 shows how Ruby creates a singleton class when executing Listing 9-4. Ruby evaluates the expression provided as a prefix to `def`: `some_quote`. Because `some_quote` is an object instance, Ruby creates a new singleton class for `some_quote` and then assigns the new method to this singleton class. Using the `def` keyword with an object prefix instructs Ruby either to use a metaclass (if the prefix is a class) or to create a singleton class (if the prefix is some other object).

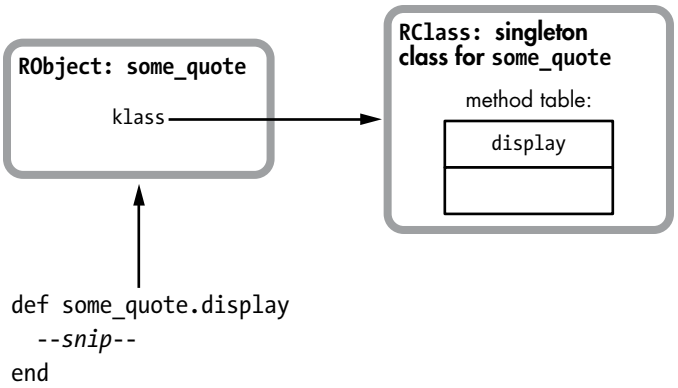

Figure 9-6: Providing an object prefix to def instructs Ruby to add the new method to the object's singleton class.

Defining Methods Using Singleton Classes in a Lexical Scope

You can also declare a new lexical scope for adding methods to a single object instance using the `class <<` syntax, as shown in Listing 9-5.

```
class Quote
end

some_quote = Quote.new
❶ class << some_quote
    def display
      puts "The quick brown fox jumped over the lazy dog."
    end
end
```

Listing 9-5: Adding a singleton method using `class <<`

The difference between this code and that in Listing 9-4 appears at ❶, when we use the `class <<` syntax with the expression `some_quote`, which evaluates to a single object instance. As shown in Figure 9-7, `class << some_quote` instructs Ruby to create a new singleton class along with a new lexical scope.

On the left side of Figure 9-7, you can see some of the code from Listing 9-5. Ruby first evaluates the expression `some_quote` and finds it is an object, not a class. Figure 9-7 indicates this with the long arrow pointing to the `RObject` structure for `some_quote`. Because it is not a class, Ruby creates a new singleton class for `some_quote` and also creates a new lexical scope. Next, it sets the class for the new scope to be the new singleton class. If a singleton class for `some_quote` already exists, Ruby will reuse it.

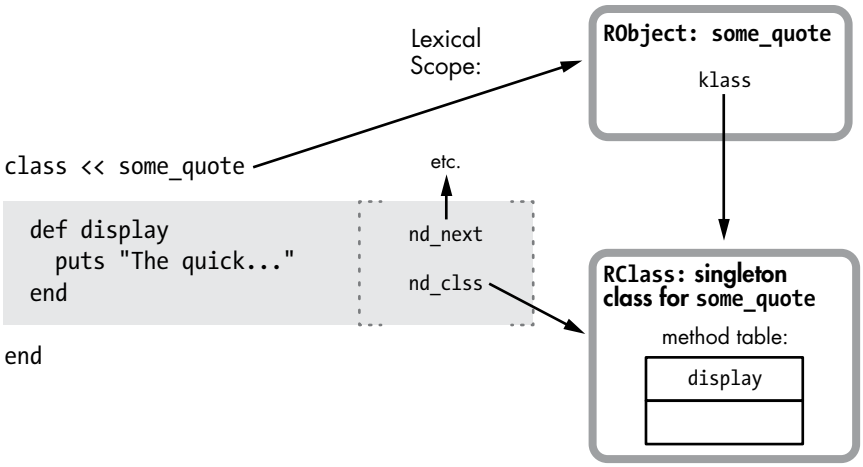

Figure 9-7: Ruby creates a new singleton class and lexical scope for `some_quote`.

Creating Refinements

Ruby 2.0's *refinements* feature gave us the ability to define methods and add them to a class later if we wish. To see how this works, we'll use the same `Quote` class and `display` method we used in Listing 9-1, repeated here for convenience.

```
class Quote
  def display
    puts "The quick brown fox jumped over the lazy dog."
  end
end
```

Now suppose elsewhere in our Ruby application we want to override or change what `display` does without changing the `Quote` class everywhere. Ruby provides an elegant way to do this, as shown in Listing 9-6.

```
module AllCaps
  refine Quote do
    def display
      puts "THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG."
    end
  end
end
```

Listing 9-6: Refining a class inside a module

In `refine Quote do`, we use the `refine` method and pass the `Quote` class as a parameter. This defines new behavior for `Quote` that we can activate later. Figure 9-8 shows what happens internally when we call `refine`.

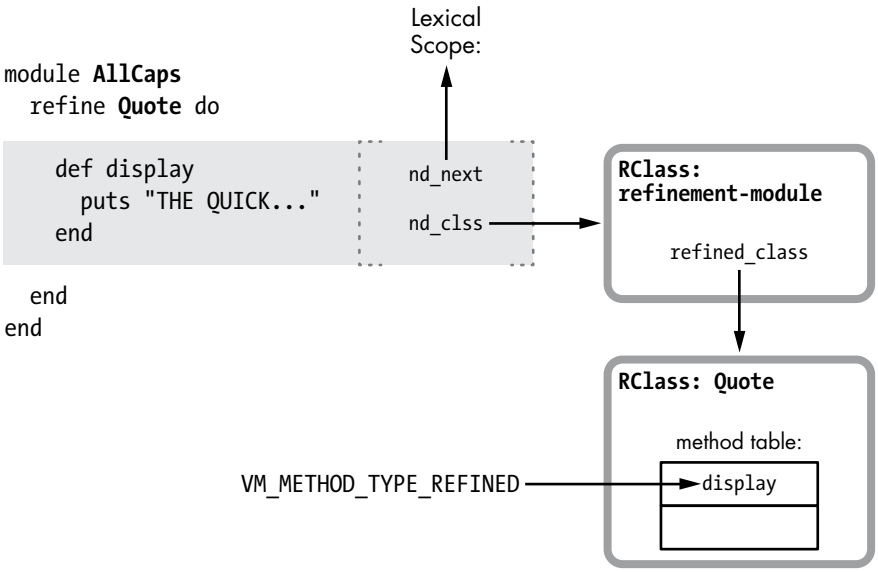

Figure 9-8: Ruby creates a special module when you call `refine` and updates the type of the target class's methods.

Working our way through Figure 9-8 from the top-left corner down, we see the following:

- The `refine` method creates a new lexical scope (the shaded rectangle).
- Ruby creates a new "refinement" module and uses that as the class for this new scope.
- Ruby saves a pointer to the `Quote` class in `refined_class` inside the new refinement module.

As you define new methods in the refine block, Ruby saves them in the refinement module. But it also follows the `refined_class` pointer and updates the same methods in the target class to use the method type `VM_METHOD_TYPE_REFINED`.

Using Refinements

You can decide to activate these "refined" methods in a specific part of your program with the `using` method, as shown in Listing 9-7.

```
❶ Quote.new.display
   => The quick brown...

❷ using AllCaps

❸ Quote.new.display
   => THE QUICK BROWN...
```

Listing 9-7: Activating a refined method

When we first call display at ❶, Ruby uses the original method. Then, at ❷ we activate the refinement with `using`, which causes Ruby to use the updated method when we call display again at ❸.

The `using` method attaches the refinements from the specified module to the current lexical scope. As I write this, the current version of Ruby, 2.0, allows you to use refinements only in the top-level scope, as in this example; `using` is a method of the top-level `main` object. (Future versions may allow you to use refinements in any lexical scope in your program.) Figure 9-9 shows how Ruby internally associates the refinement with the top-level lexical scope.

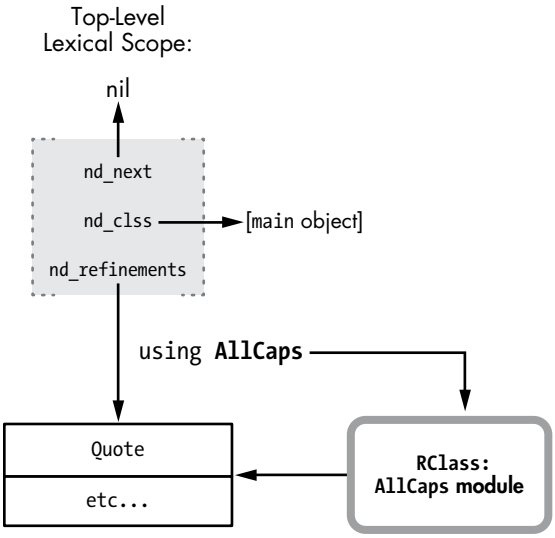

Figure 9-9: The `using` method associates a module's refinements with the top-level lexical scope.

Notice how each lexical scope contains an `nd_refinements` pointer, which tracks the refinements active in that scope. The `using` method sets `nd_refinements`, which would otherwise be `nil`.

And finally, Figure 9-10 shows how Ruby's method dispatch algorithm finds the updated method when I call it.

Ruby uses a complex method dispatch process when you call methods. One portion of this algorithm looks for `VM_METHOD_TYPE_REFINED` methods. When it encounters a refined method, Ruby looks in the current lexical scope for any active refinements. If it finds an active refinement, Ruby calls the refined method; otherwise, it calls the original method.

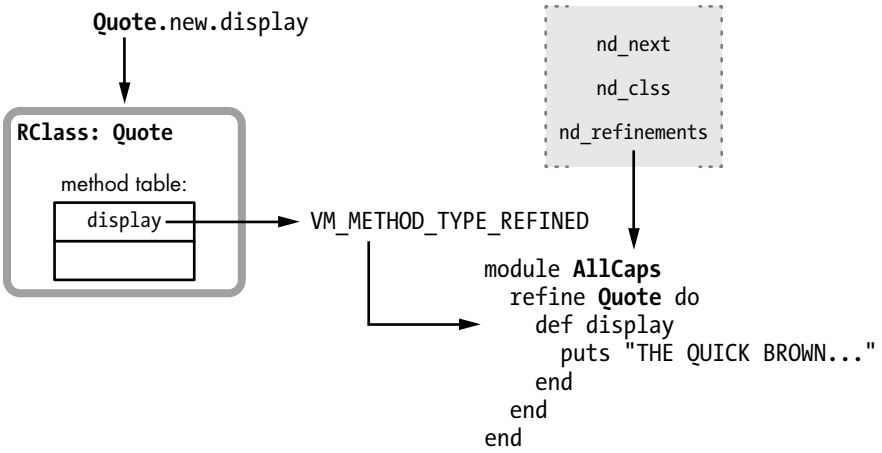

Figure 9-10: Ruby looks for a method in the refine block when the original method is marked with `VM_METHOD_TYPE_REFINED`.

Experiment 9-1: Who Am I? How self Changes with Lexical Scope

We've seen various ways to define methods in Ruby. We created methods in the usual way using the `def` keyword. Then, we looked at how to create methods on a metaclass and on a singleton class and how to use refinements.

While each technique adds the method to a different class, each also follows a simple rule: Ruby adds the new method to the class corresponding to the current lexical scope for each technique. (The `def` keyword, however, assigns the method to a different class when you use a prefix.) With refinements, the current scope's class is actually the special module created to hold the refined methods. In fact, this is one of the important roles lexical scope plays in Ruby: It identifies which class or module we are currently adding methods to.

We also know that the `self` keyword returns the current object—the receiver of the method currently being executed by Ruby. Recall that YARV saves the current value of `self` for each level of your Ruby call stack in the `rb_control_frame_t` structure. Is this object the same as the class for the current lexical scope?

self in the Top Scope

Let's see how the value of `self` changes as we run a simple program beginning with Listing 9-8.

```
p self
 => main
p Module.nesting
 => []
```

Listing 9-8: A simple Ruby program with only one lexical scope

To keep things simple, I've shown the output from the console inline. You can see that Ruby creates a `top self` object before it starts to execute your code. This object serves as the receiver for method calls in the top-level scope. Ruby represents this object with the string `main`.

The `Module.nesting` call returns an array showing the lexical scope stack—that is, which modules are "nested" until that point in the code. This array will contain an element for each lexical scope in the lexical scope stack. Because we're at the top level of the script, Ruby returns an empty array.

Figure 9-11 shows the lexical scope stack and the value `self` for this simple program.

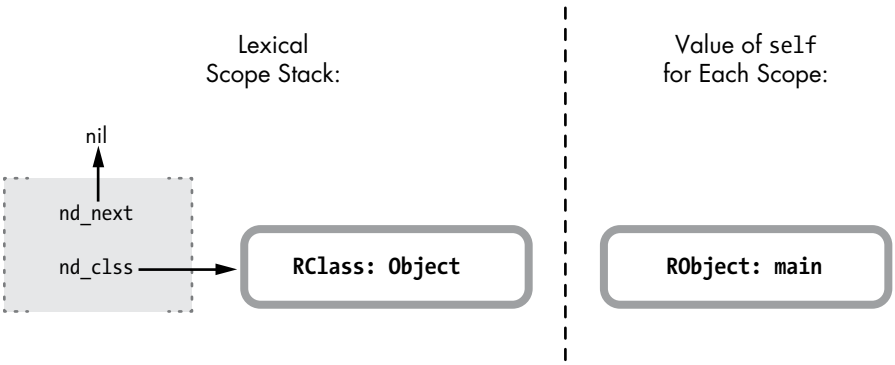

Figure 9-11: At the top level, Ruby sets `self` to the `main` object and has a single entry in the lexical scope stack.

On the right of this figure, you see the `main` object: the current value of `self`. On the left side is the lexical scope stack, which contains just a single entry for the top-level scope. Ruby sets the class of the top scope to the class of the `main` object, which is the `Object` class.

NOTE *Recall when you declare a new method using the `def` keyword, Ruby adds the method to the class for the current lexical scope. We've just seen the class for the top-level lexical scope is `Object`. Therefore, we can conclude that when you define a method at the top level of your script, outside of any class or module, Ruby adds the method to the `Object` class. You can call methods you define at the top level from anywhere because `Object` is a superclass of every other class.*

self in a Class Scope

Now let's define a new class and see what happens to the value of `self` and the lexical scope stack, as shown in Listing 9-9.

```
p self
p Module.nesting

class Quote
  p self
❶   => Quote
```

```
  p Module.nesting
❷  => [Quote]
end
```

Listing 9-9: Declaring a new class changes `self` and creates a new entry in the lexical scope stack.

The output from the print statements is shown inline. We see at ❶ that Ruby has changed `self` to `Quote`—the new class—and we see at ❷ that there's a new level added to the lexical scope stack. Figure 9-12 shows a summary.

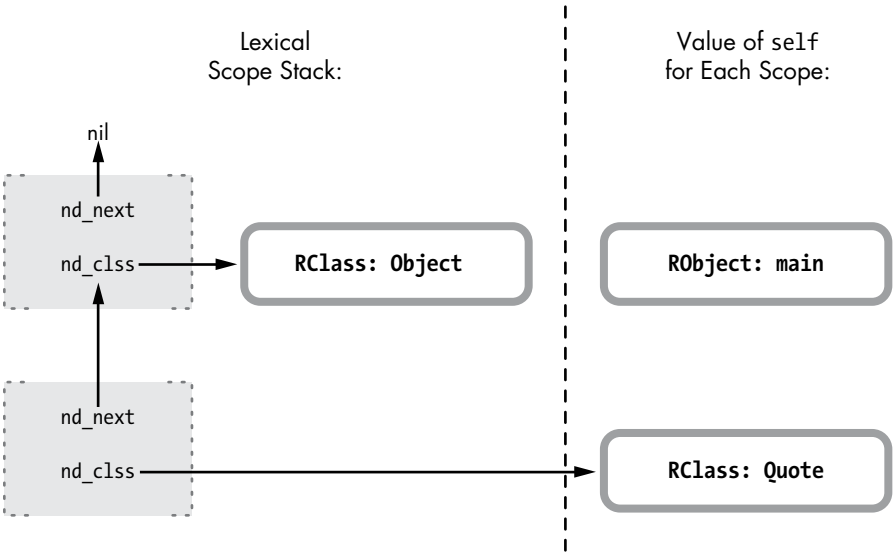

Figure 9-12: Now `self` is the same as the class for the current lexical scope.

On the left side of this figure, we see the lexical scope stack. The top scope is on the top left, and under it we see the new lexical scope created by the `class` keyword. Meanwhile, on the right side of the figure, we see how the value of `self` changes when we call `class`. On the top level, `self` was set to the `main` object, but when we call `class`, Ruby changes `self` to the new class.

self in a Metaclass Scope

Let's use the `class << self` syntax to create a new metaclass scope. Listing 9-10 shows the same program with a few more lines of code.

```
p self
p Module.nesting

class Quote
  p self
  p Module.nesting
```

```
  class << self
    p self
❶    => #<Class:Quote>
    p Module.nesting
❷    => [#<Class:Quote>, Quote]
  end
end
```

Listing 9-10: Declaring a metaclass scope

At ❶ we see that Ruby has changed the value of `self` again. The syntax `#<Class:Quote>` indicates that `self` was set to `Quote`'s metaclass. At ❷ we see that Ruby has also added another level to the lexical scope stack. Figure 9-13 shows the next level in the stack.

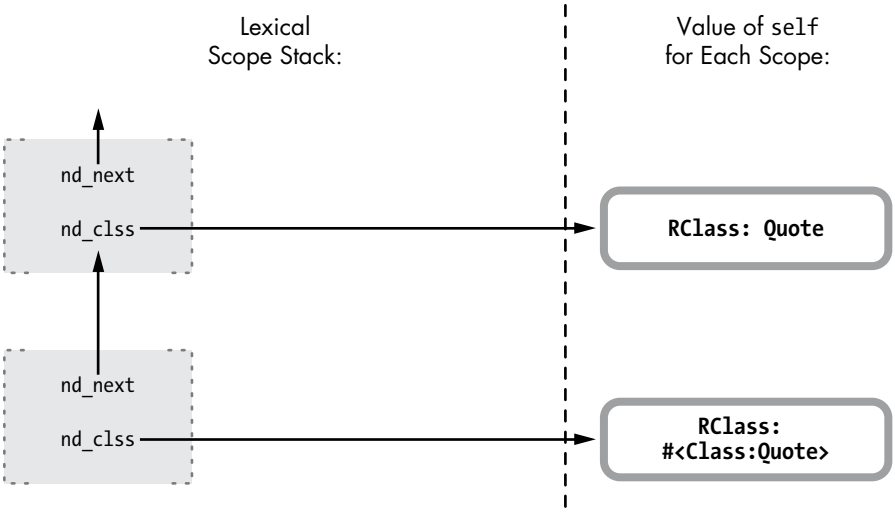

Figure 9-13: A new lexical scope is created for the metaclass.

On the left, we can see that Ruby created a new scope when it executed `class << self`. The right side of the figure shows the value of `self` in the new scope, the metaclass for `Quote`.

self Inside a Class Method

Now for one more test. Suppose we add a class method to the `Quote` class and then call it as shown in Listing 9-11. (The output is at the bottom because the `p` statements aren't called until we call `class_method`.)

```
p self
p Module.nesting

class Quote
  p self
  p Module.nesting
```

```
  class << self
    p self
    p Module.nesting

    def class_method
      p self
      p Module.nesting
    end
  end
end

Quote.class_method
❶ => Quote
❷ => [#<Class:Quote>, Quote]
```

Listing 9-11: Declaring and calling a class method

At ❶ we see that Ruby sets `self` back to the `Quote` class when we call `class_method`. This makes sense: When we call a method on a receiver, Ruby always sets `self` to be the receiver. Because we call a class method in this case, Ruby sets the receiver to that class.

At ❷ we see that Ruby hasn't changed the lexical scope stack. It's still set to `[#<Class:Quote>, Quote]`, as shown in Figure 9-14.

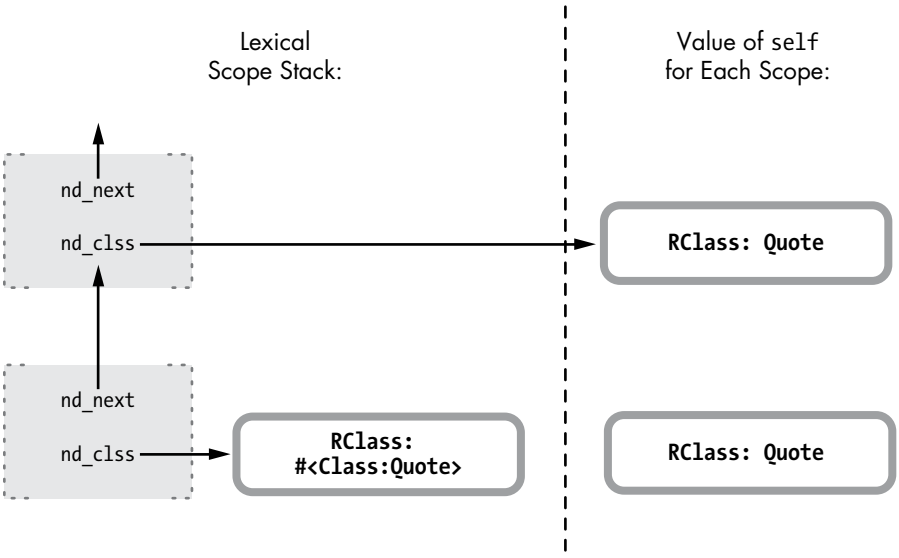

Figure 9-14: When you call a method, Ruby changes `self` but doesn't create a new scope.

Notice that the lexical scope hasn't changed but `self` has been changed to `Quote`, the receiver of the method call.

You can use these general rules to keep track of `self` and lexical scope:

- Inside a class or module scope, `self` will always be set to that class or module. Ruby creates a new lexical scope when you use the `class` or `module` keywords and sets the class for that scope to the new class or module.
- Inside a method (including a class method), Ruby will set `self` to the receiver of that method call.

Metaprogramming and Closures: eval, instance_eval, and binding

In Chapter 8 we learned that blocks are Ruby's implementation of closures, and we saw how blocks bring together a function with the environment where that function was referenced. In Ruby, metaprogramming and closures are closely related. Many of Ruby's metaprogramming constructs also act as closures, giving the code inside them access to the referencing environment. We'll learn about three important metaprogramming features and how each gives you access to the referencing environment by acting as a closure in just the way blocks do.

Code That Writes Code

In Ruby, the `eval` method is metaprogramming in its purest form: You pass a string to `eval`, and Ruby immediately parses, compiles, and executes the code, as shown in Listing 9-12.

```
str = "puts"
str += " 2"
str += " +"
str += " 2"
eval(str)
```

Listing 9-12: Parsing and compiling code using `eval`

We dynamically construct the string `puts 2+2` and pass it to `eval`. Ruby then evaluates the string. That is, it tokenizes, parses, and compiles it using the same Bison grammar rules and parse engine that it did when it first processed the primary Ruby script. Once this process is finished and Ruby has another new set of YARV bytecode instructions, it executes the new code.

But one very important detail about `eval` isn't obvious in Listing 9-12. Specifically, Ruby evaluates the new code string in the same context from where you called `eval`. To see what I mean, look at Listing 9-13.

```
a = 2
b = 3
str = "puts"
str += " a"
str += " +"
```

```
str += " b"
❶ eval(str)
```

Listing 9-13: It isn't obvious here, but `eval` accesses the surrounding scope via a closure, too.

You would expect the result from running this code to be 5, but notice the difference between Listings 9-12 and 9-13. Listing 9-13 refers to the local variables `a` and `b` from the surrounding scope, and Ruby can access their values. Figure 9-15 shows how YARV's internal stack looks just before calling `eval` at ❶.

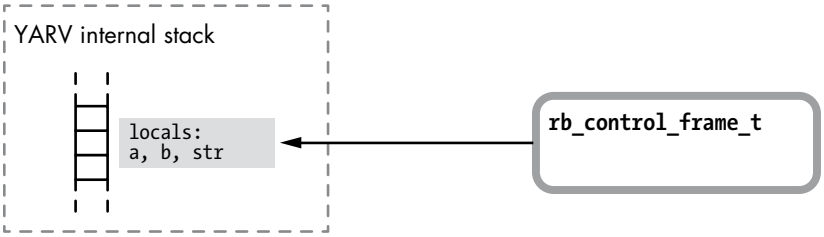

Figure 9-15: Ruby saves the local variables `a`, `b`, and `str` on YARV's internal stack as usual.

As expected, we see that Ruby has saved the values of `a`, `b`, and `str` on the stack to the left. On the right, we have the `rb_control_frame_t` structure, which represents the outer, or main, scope of this script.

Figure 9-16 shows what happens when we call the `eval` method.

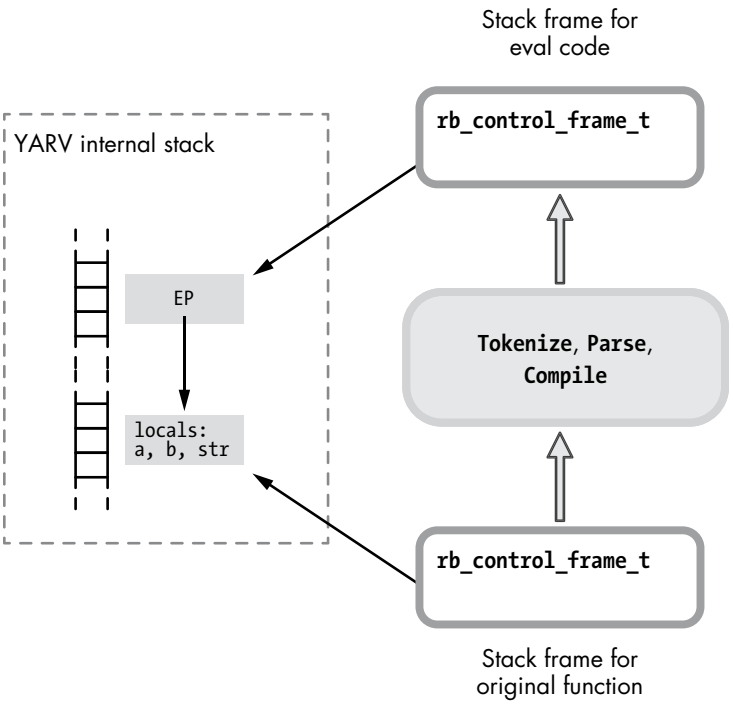

Figure 9-16: Calling `eval` and accessing values from the parent scope

Calling `eval` invokes the parser and compiler on the text we pass it. When the compiler finishes, Ruby creates a new stack frame (`rb_control_frame_t`) for use in running the new compiled code (as shown at the top). Notice, however, that Ruby sets the `EP` in this new stack frame to point to the lower stack frame where the variables `a` and `b` are. This pointer allows the code passed to `eval` to access these values.

Ruby's use of `EP` here should look familiar. Aside from parsing and compiling the code dynamically, `eval` works the same way as if we had passed a block to some function, as in Listing 9-14.

```
a = 2
b = 3
10.times do
  puts a+b
end
```

Listing 9-14: Code inside a block can access variables from the surrounding scope.

In other words, the `eval` method creates a closure: the combination of a function and the environment where that function was referenced. In this case, the function is the newly compiled code, and the environment is where we call `eval` from.

Calling eval with binding

The `eval` method can take a second parameter: a *binding*. A binding is a closure without a function—that is, it's just the referencing environment. Think of bindings as a pointer to a YARV stack frame. Passing a binding value to Ruby indicates that you don't want to use the current context as the closure's environment but instead want to use some other referencing environment. Listing 9-15 shows an example.

```
def get_binding
  a = 2
  b = 3
❶ binding
end
❷ eval("puts a+b", get_binding)
```

Listing 9-15: Using `binding` to access variables from some other environment

The function `get_binding` contains the local variables `a` and `b`, but it also returns a binding at ❶. At the bottom of the listing, we again want Ruby to dynamically compile and execute the code string and print out the result. By passing the binding returned by `get_binding` to `eval`, we tell Ruby to evaluate `puts a+b` in the context of the `get_binding` function. If we had called `eval` without the binding, it would have created new, empty local variables `a` and `b`.

Ruby makes a persistent copy of this environment in the heap because you might call `eval` long after the current frame has been popped off the

stack. Even though `get_binding` has already returned in this example, Ruby can still access the values of `a` and `b` when it executes the code parsed and compiled by `eval` at ❷.

Figure 9-17 shows what happens internally when we call `binding`.

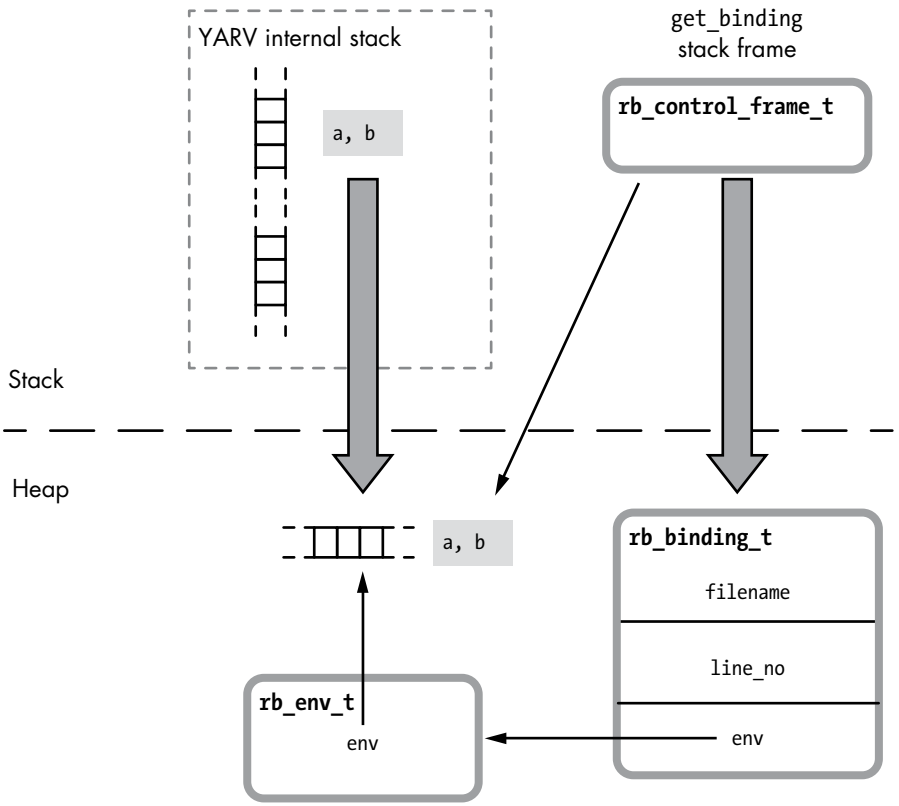

Figure 9-17: Calling `binding` saves a copy of the current stack frame in the heap.

This figure resembles what Ruby does when you call `lambda` (see Figure 8-18 on page 210), except that Ruby creates an `rb_binding_t` C structure instead of an `rb_proc_t` structure. The binding structure is simply a wrapper around the internal environment structure—the heap copy of the stack frame. The binding structure also contains the file name and line number of the location from where you called `binding`.

As with the proc object, Ruby uses the `RTypedData` structure to wrap a Ruby object around the `rb_binding_t` C structure (see Figure 9-18).

The binding object allows you to create a closure and then obtain and treat its environment as a data value. However, the closure created by the binding doesn't contain any code; it has no function. You might think of the binding object as an indirect way to access, save, and pass around Ruby's internal `rb_env_t` structure.

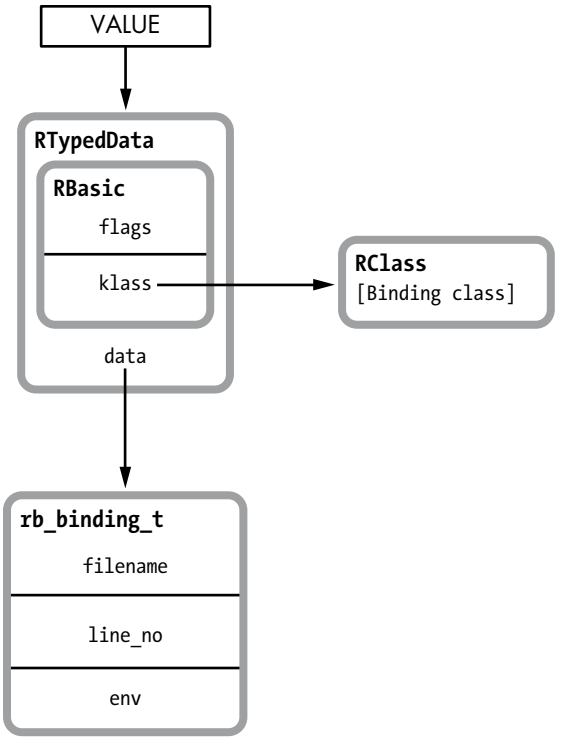

Figure 9-18: Ruby uses `RTypedData` to wrap a Ruby object around the `rb_binding_t` structure.

An instance_eval Example

Now for a variation on the `eval` method: `instance_eval` is shown in action in Listing 9-16.

```
❶ class Quote
    def initialize
❷     @str = "The quick brown fox"
    end
  end
  str2 = "jumps over the lazy dog."
❸ obj = Quote.new
❹ obj.instance_eval do
❺   puts "#{@str} #{str2}"
  end
```

Listing 9-16: The code inside `instance_eval` has access to `obj`'s instance variable.

Here's what's going on:

- At ❶ we create a Ruby class called `Quote` that saves the first half of the string in an instance variable in `initialize` at ❷.

- At ❸ we create an instance of the `Quote` class and then call `instance_eval` at ❹, passing a block. The `instance_eval` method is similar to `eval`, except that it evaluates the given string in the context of the receiver, or the object we call it on. As shown here, we can pass a block to `instance_eval` instead of a string if we don't want to dynamically parse and compile code.
- The block we pass to `instance_eval` prints out the string at ❺, accessing the first half of the string from the `obj`'s instance variable and the second half from the surrounding scope, or environment.

How can this possibly work? It seems that the block passed to `instance_eval` has two environments: the quote instance and the surrounding code environment. In other words, the `@str` variable comes from one place and `str2` from another.

Another Important Part of Ruby Closures

This example highlights another important part of closure environments in Ruby: the current value of `self`. Recall that the `rb_control_frame_t` structure for each stack frame, or level, in your Ruby call stack contains a `self` pointer, along with the `PC`, `SP`, and `EP` pointers and other values (see Figure 9-19).

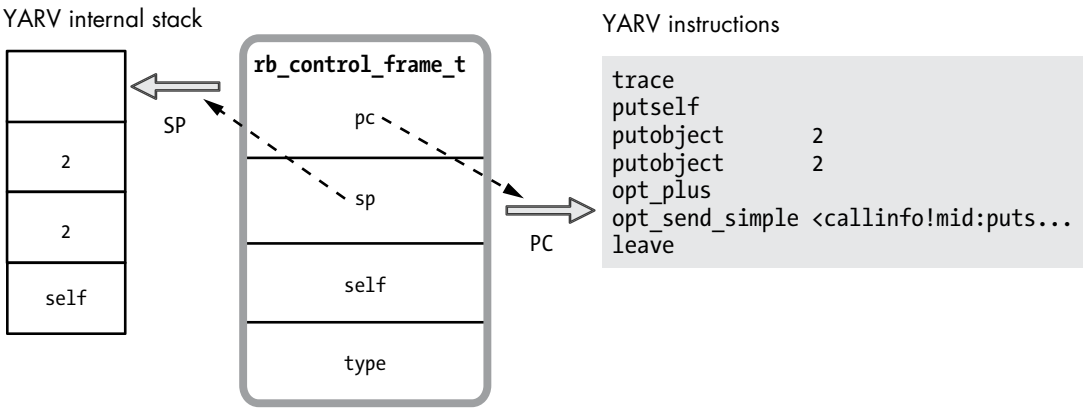

Figure 9-19: The `rb_control_frame_t` structure

The `self` pointer records the current value of `self` in your Ruby project; it indicates which object is the owner of the method Ruby is currently executing at that time. Each level in your Ruby call stack can contain a different value for `self`.

Recall that whenever you create a closure Ruby sets the `EP`, or environment pointer, in the `rb_block_t` structure to the referencing environment, giving the code inside the block access to the surrounding variables. And, as it turns out, Ruby also copies the value of `self` into `rb_block_t`. This means that the current object is also a part of closures in Ruby. Figure 9-20 looks at what closures contain in Ruby.

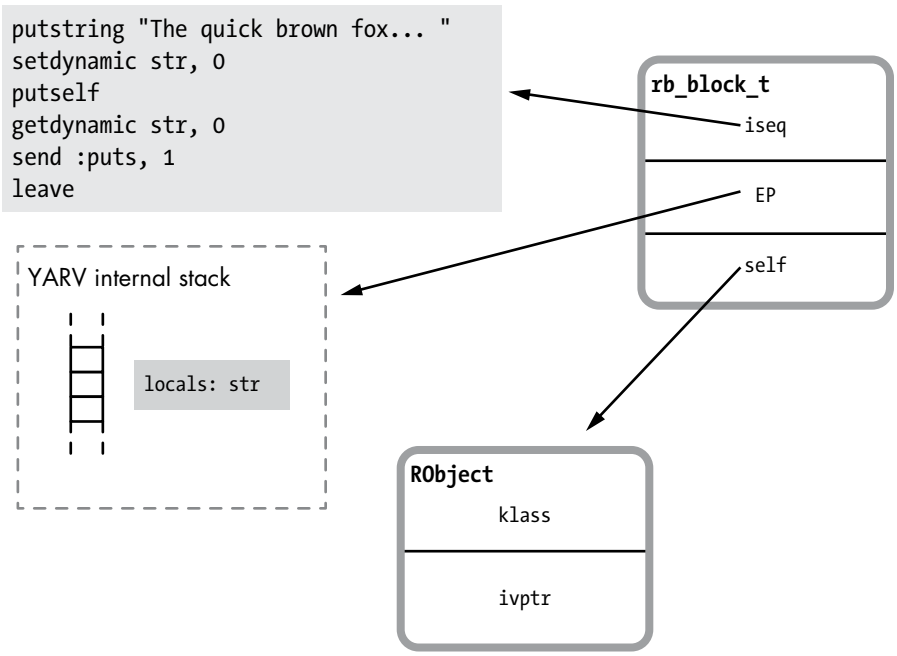

Figure 9-20: In Ruby, closure environments include both the stack frame and the current object from the referencing code.

Because the `rb_block_t` structure contains the value of `self` from the referencing environment, code inside a block can access the values and methods of the object that was active when the closure was created or referenced. This ability probably seems obvious for a block: The current object before and after you call a block doesn't change. However, if you use a lambda, proc, or binding, Ruby will remember what the current object was when you created it. And, as we'll see shortly with `instance_eval`, Ruby can sometimes change `self` when you create a closure, giving your code access to a different object's values and methods.

instance_eval Changes self to the Receiver

When you call `instance_eval` at ❹ in Listing 9-16, Ruby creates both a closure and a new lexical scope. For example, as you can see in Figure 9-21, the new stack frame for the code inside `instance_eval` uses new values for both `EP` and `self`.

On the left of the figure, we see that executing `instance_eval` creates a closure. This result should be no surprise. Passing a block to `instance_eval` at ❹ in Listing 9-16 creates a new level on the stack and sets `EP` to the referencing environment, giving the code inside the block access to the variables `str2` and `obj`.

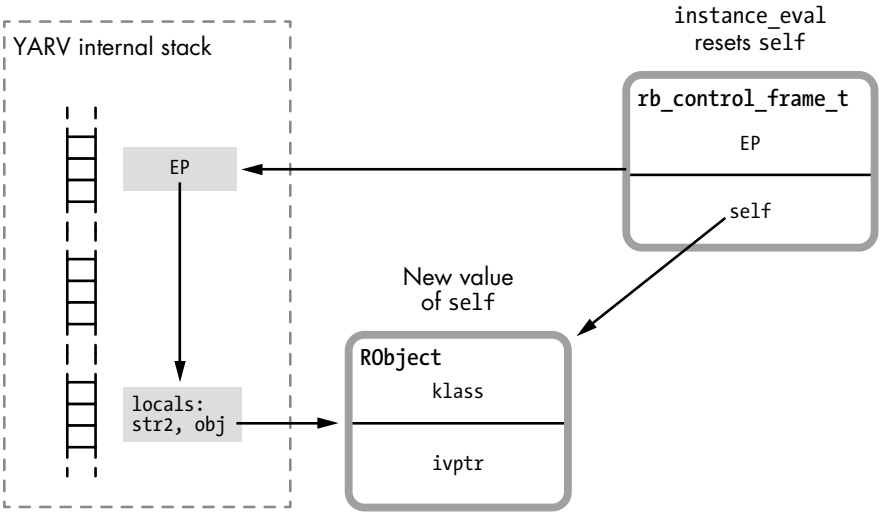

Figure 9-21: The stack frame created by running `instance_eval` has a new value for `self`.

However, as you can see on the right of the figure, `instance_eval` also changes the value of `self` in the new closure. When the code inside the `instance_eval` block runs, `self` points to the receiver of `instance_eval`, or `obj`, in Listing 9-16. This allows the code inside `instance_eval` to access the values inside the receiver. In Listing 9-16, the code at ❺ can access both `@str` from inside `obj` and `str2` from the surrounding code.

instance_eval Creates a Singleton Class for a New Lexical Scope

The `instance_eval` method also creates a new singleton class and sets it as the class for a new lexical scope, as shown in Figure 9-22.

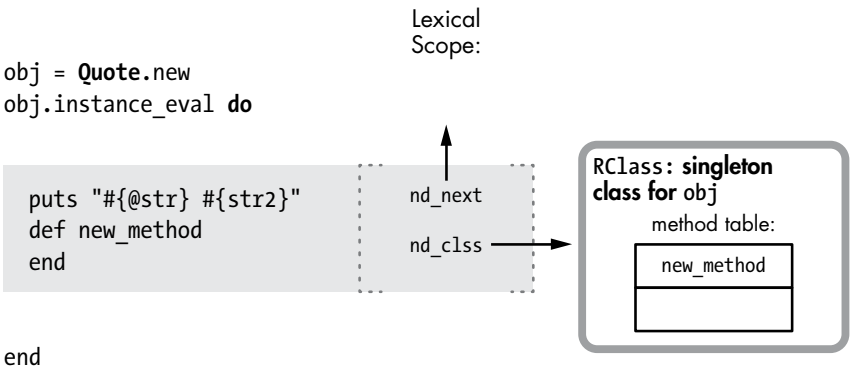

Figure 9-22: `instance_eval` creates a lexical scope for a new singleton class.

While executing `instance_eval`, Ruby creates a new lexical scope, as shown by the shaded rectangle inside the `instance_eval` block. If we had passed a string to `instance_eval`, Ruby would have parsed and compiled the string and then created a new lexical scope in the same way.

Along with the new lexical scope, Ruby creates a singleton class for the receiver, `obj`. The singleton class allows you to define new methods for the receiver object (see Figure 9-22): The `def new_method` call inside the `instance_eval` block adds `new_method` to the singleton class for `obj`. As a singleton class, `obj` will have the new method, but no other objects or classes in the program will have access to it. (The metaprogramming methods `class_eval` and `module_eval` work in a similar way and also create a new lexical scope; however, they just use the target class or module for the new scope and don't create a metaclass or singleton class.)

HOW RUBY KEEPS TRACK OF LEXICAL SCOPE FOR BLOCKS

Let's take a closer look at how Ruby represents lexical scopes internally. Figure 9-23 shows the lexical scope Ruby creates for the `Quote` class.

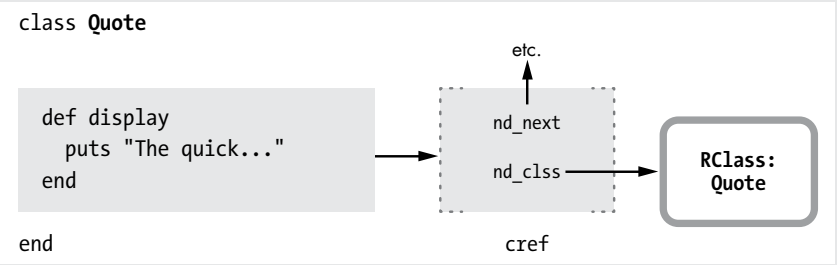

Figure 9-23: Ruby's C source code internally uses a separate structure called `cref` to track lexical scopes.

You can see the `display` method's code snippet represented as a rectangle on the left side of the figure, inside the class `Quote` declaration. On the right side of the rectangle, you can see a small arrow pointing to a structure labeled `cref`, which is the actual lexical scope. This, in turn, contains a pointer to the `Quote` class (`nd_clss`) and to the parent lexical scope (`nd_next`).

As indicated by the figure, Ruby's C source code internally represents lexical scopes using these `cref` structures. The small arrow on the left shows that each piece of code in your program refers to a `cref` structure with a pointer. This pointer keeps track of which lexical scope that piece of code belongs to.

Notice one important detail about Figure 9-23: Both the code snippet and lexical scope inside the `class Quote` declaration refer to a single `RClass` structure. There's a one-to-one correspondence between code, lexical scope, and class. Every time Ruby executes the code inside the `class Quote` declaration, it uses the same copy of the `RClass` structure, the one for `Quote`. This behavior seems obvious; the code inside a class declaration always refers to the same class.

For blocks, however, things aren't so simple. Using metaprogramming methods such as `instance_eval`, you can specify a different lexical scope for the same piece of code—a block, for example—to use each time it is executed. Figure 9-24 shows the problem.

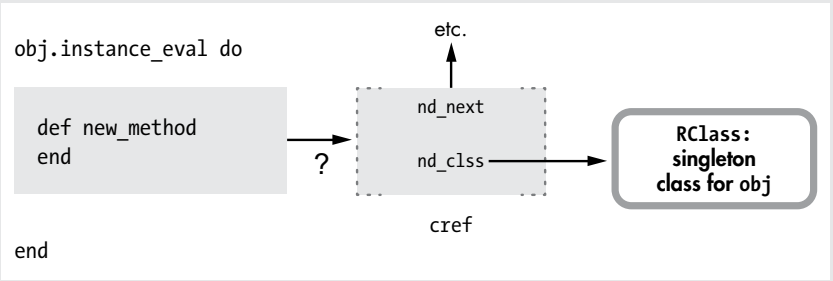

Figure 9-24: The block's code can't refer to a single lexical scope because the scope's class depends on the value of `obj`.

We learned in the previous section that Ruby creates a singleton class for the lexical scope created by `instance_eval`. However, this code might be run many times for different values of `obj`. In fact, your program might execute this code at the same time in different threads. This requirement means that Ruby can't keep a pointer to a single `cref` structure for the block as it does for a class definition. This block scope will refer to different classes at different times.

Ruby solves this problem by saving a pointer to the lexical scope used by blocks in a different place: as an entry on YARV's internal stack (see Figure 9-25).

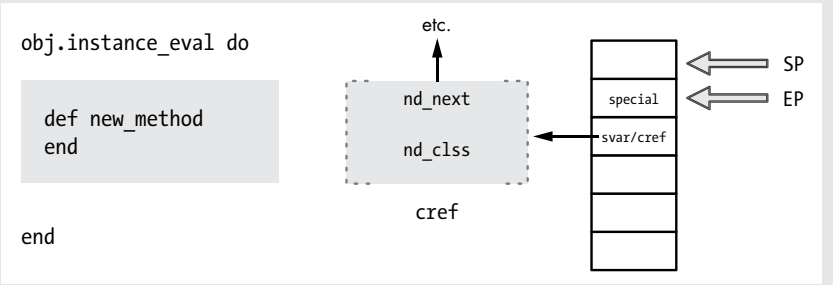

Figure 9-25: Ruby tracks lexical scope for blocks using the `svar/cref` entry on the stack, not using the block's code snippet.

On the left side of the figure, you can see the call to `instance_eval` and the code snippet for the block inside. In the center of the figure is the `cref` structure for the lexical scope. On the right side, you can see YARV saves a pointer to the scope in the second entry on its stack, labeled `svar/cref`.

Recall from Chapter 3 that the second entry on YARV's internal stack contains one of two values: `svar` or `cref`. As we saw in Experiment 3-2 on page 75, `svar` saves a pointer to a table of special variables, such as the result of the last regular expression match, while executing a method. But while executing a block, YARV saves the `cref` value here instead. Usually this value isn't important because blocks normally use the lexical scope of the surrounding code. But when executing `instance_eval` and a few other metaprogramming features, such as `module_eval` and `instance_exec`, Ruby sets `cref` in this way to the current lexical scope.

Experiment 9-2: Using a Closure to Define a Method

Another common metaprogramming pattern in Ruby is to dynamically define methods in a class using `define_method`. For example, Listing 9-17 shows a simple Ruby class that prints out a string when you call `display`.

```
class Quote
  def initialize
    @str = "The quick brown fox jumps over the lazy dog"
  end
  def display
    puts @str
  end
end
Quote.new.display
 => The quick brown fox jumps over the lazy dog
```

Listing 9-17: A Ruby class that displays a string from an instance variable

This code is similar to that in Listing 9-1, except that we use an instance variable `@str` to hold the string value.

Using define_method

We could have used metaprogramming to define `display` in a more verbose but dynamic way, as shown in Listing 9-18.

```
class Quote
  def initialize
    @str = "The quick brown fox jumps over the lazy dog"
  end
❶ define_method :display do
    puts @str
  end
end
```

Listing 9-18: Using `define_method` to create a method

We call `define_method` at ❶ instead of the normal `def` keyword. Because the name of the new method is passed as the argument `:display`, we can dynamically construct the method name from some data values or iterate over an array of method names, calling `define_method` for each one.

But there is another subtle difference between `def` and `define_method`. For `define_method` we provide the body of the method as a block; that is, we use a `do` keyword at ❶. This syntax difference may seem minor, but remember that blocks are actually closures. Adding `do` introduces a closure, meaning that the code inside the new method has access to the environment outside. This is not the case with the `def` keyword.

There are no local variables present in Listing 9-18 when we call `define_method`, but suppose that another place in our application did have

values that we wanted to use inside our new method. By using a closure, Ruby makes an internal copy of the surrounding environment on the heap, which the new method will be able to access.

Methods Acting as Closures

Now for another test. Listing 9-19 stores only the first half of the string in the instance variable. In a moment, we'll write a new method for the `Quote` class to access this.

```
class Quote
  def initialize
    @str = "The quick brown fox"
  end
end
```

Listing 9-19: Now `@str` has only the first half of the string.

Listing 9-20 shows how we can use a closure to access both the instance variable and the surrounding environment.

```
def create_method_using_a_closure
  str2 = "jumps over the lazy dog."
❶ Quote.send(:define_method, :display) do
    puts "#{@str} #{str2}"
  end
end
```

Listing 9-20: Using a closure with `define_method`

Because `define_method` is a private method in the `Module` class, we need to use the confusing `send` syntax at ❶. Earlier, at ❶ in Listing 9-18, we were able to call `define_method` directly because we used it inside the class's scope. We can't do that from other places in the application. By using `send`, the `create_method_using_a_closure` method can call a private method that it wouldn't normally have had access to.

But more importantly, notice that the `str2` variable is preserved in the heap for the new method to use even after `create_method_using_a_closure` returns:

```
create_method_using_a_closure
Quote.new.display
 => The quick brown fox jumps over the lazy dog.
```

Internally, Ruby treats this as a call to `lambda`. That is, this code functions the same way as if I had written the code in Listing 9-21.

```
class Quote
  def initialize
    @str = "The quick brown fox"
  end
```

```
  end
  def create_method_using_a_closure
    str2 = "jumps over the lazy dog."
    lambda do
      puts "#{@str} #{str2}"
    end
  end
❶ Quote.send(:define_method, :display, create_method_using_a_closure)
❷ Quote.new.display
```

Listing 9-21: Passing a proc to `define_method`

Listing 9-21 separates the code that creates the closure and defines the method. Because at ❶ we pass three arguments to `define_method`, Ruby expects the third to be a proc object. While this is an even more verbose way to write this code, it's a bit less confusing because calling `lambda` makes it clear that Ruby will create a closure.

Finally, when we call the `new` method at ❷, Ruby resets the `self` pointer from the closure to receiver object, similar to the way that `instance_eval` works. This allows the new method to access `@str` as you would expect.

Summary

In this chapter we've seen how the concept of closures—the idea central to the way blocks, lambdas, and procs work in Ruby—also applies to methods such as `eval`, `instance_eval`, and `define_method`. The same underlying concept explains how these different Ruby methods work. In a similar way, the concept of lexical scope underpins all of the ways that Ruby allows you to create a method and assign it to a class. Understanding the concept of lexical scope should make the different uses of Ruby's `def` keyword and `class <<` syntax easier to understand.

While metaprogramming might seem complex at first, learning how Ruby works internally can help us understand what Ruby's metaprogramming features actually do. What seems initially like a large set of different, unrelated methods in a confusing API turn out to be related by a few important ideas. Studying Ruby internals allows us to see these concepts and to understand what they mean.