Chapter 24

Metaprogramming

The Jacquard loom, invented more than 200 years ago, was the first device controlled using punched cards—rows of holes in each card were used to control the pattern woven into the cloth. But imagine if instead of churning out fabric, the loom could punch more cards, and those cards could be fed back into the mechanism. The machine could be used to create new programming that it could then execute. And that would be metaprogramming—writing code that writes code.

Programming is all about building layers of abstractions. As you solve problems, you're building bridges from the unrelenting and mechanical world of silicon to the more ambiguous and fluid world we inhabit. Some programming languages—such as C—are close to the machine. The distance from C code to the application domain can be large. Other languages—Ruby, perhaps—provide higher-level abstractions and hence let you start coding closer to the target domain. For this reason, most people consider a higher-level language to be a better starting place for application development (although they'll argue about the choice of language).

But when you metaprogram, you are no longer limited to the set of abstractions built in to your programming language. Instead, you can create new abstractions that are integrated into the host language. In effect, you're creating a new, domain-specific programming language—one that's designed to let you express the concepts you need to solve your particular problem.

Ruby makes metaprogramming easy. As a result, most advanced Ruby programmers will use metaprogramming techniques to simplify their code. This chapter shows how they do it. It isn't intended to be an exhaustive survey of metaprogramming techniques. Instead, we'll look at the underlying Ruby principles that make metaprogramming possible. From there you'll be able to invent your own metaprogramming idioms.

Objects and Classes

Classes and objects are obviously central to Ruby, but at first sight they can be a little confusing. There seem to be a lot of concepts: classes, objects, class objects, instance methods, class methods, singleton classes, and virtual classes. In reality, however, Ruby has just a single underlying class and object structure.

A Ruby object has three components: a set of flags, some instance variables, and an associated class.

A Ruby class is itself an object of class Class. It contains all the things an object has plus a set of method definitions and a reference to a superclass (which is itself another class).

And, basically, that's it. From here, you could work out the details of metaprogramming for yourself. But, as always, the devil lurks in the details, so let's dig a little deeper.

self and Method Calling

Ruby has the concept of the *current object*. This current object is referenced by the built-in, read-only variable self. self has two significant roles in a running Ruby program.

First, self controls how Ruby finds instance variables. We already said that every object carries around a set of instance variables. When you access an instance variable, Ruby looks for it in the object referenced by self.

Second, self plays a vital role in method calling. In Ruby, each method call is made on some object. This object is called the *receiver* of the call. When you make a method call such as items.size, the object referenced by the variable items is the receiver and size is the method to invoke.

If you make a method call such as puts "hi", there's no explicit receiver. In this case, Ruby uses the current object, self, as the receiver. It goes to self's class and looks up the method (in this case, puts). If it can't find the method in the class, it looks in the class's superclass and then in that class's superclass, stopping when it runs out of superclasses (which will happen after it has looked in BasicObject).[1]

When you make a method call with an explicit receiver (for example, invoking items.size), the process is surprisingly similar. The only change—but it's a vitally important one—is the fact that self is changed for the duration of the call. Before starting the method lookup process, Ruby sets self to the receiver (the object referenced by items in this case). Then, after the call returns, Ruby restores the value that self had before the call.

Let's see how this works in practice. Here's a simple program:

Download **samples/classes_1.rb**

```
class Test
  def one
    @var = 99
    two
  end
```

1. If it can't find the method after exhausting the object's class hierarchy, Ruby looks for a method called method_missing on the original receiver, starting back at the class of self and then looking up the superclass chain.

```
  def two
    puts @var
  end
end
t = Test.new
t.one
```

produces:

```
99
```

The call to Test.new on the second-to-last line creates a new object of class Test, assigning that object to the variable t. Then, on the next line, we call the method t.one. To execute this call, Ruby sets self to t and then looks in t's class for the method one. Ruby finds the method defined on line 2 and calls it.

Inside the method, we set the instance variable @var to 99. This instance variable will be associated with the current object. What is that object? Well, the call to t.one set self to t, so within the one method, self will be that particular instance of class Test.

On the next line, the one calls the two. Because there's no explicit receiver, self is not changed. When Ruby looks for the method two, it looks in Test, the class of t.

The method two references an instance variable @var. Again, Ruby looks for this variable in the current object and finds the same variable that was set by the method one.

The call to puts at the end of two works the same way. Again, because there's no explicit receiver, self will be unchanged. Ruby looks for the puts method in the class of the current object but can't find it. It then looks in Test's superclass, class Object. Again, it doesn't find puts. However, Object mixes in the module Kernel. We'll talk more about this later, for now we can say that mixed-in modules act as if they were superclasses. The kernel module *does* define puts, so the method is found and executed.

After two and one return, Ruby resets self to the value it had before the original call to t.one.

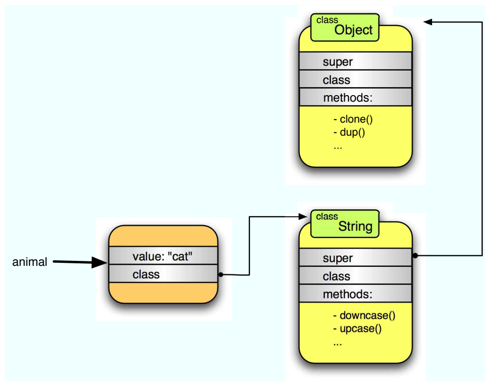
This explanation may seem labored, but understanding it is vital to mastering metaprogramming in Ruby.

self and Class Definitions

We've seen that calling a method with an explicit receiver changes self. Perhaps surprisingly, self is also changed by a class definition. This is a consequence of the fact that class definitions are actually executable code in Ruby—if we can execute code, we need to have a current object. A simple test shows what this object is:

```
class Test
  puts "In the definition of class Test"
  puts "self = #{self}"
  puts "Class of self = #{self.class}"
end
```

Figure 24.1. Object Model for a Basic Class



produces:

```
In the definition of class Test
self = Test
Class of self = Class
```

Inside a class definition, self is set to the class object of the class being defined. This means that instance variables set in a class definition will be available to class methods (because self will be the same when the variables are defined and when the methods execute):

Download **samples/classes_3.rb**

```
class Test
  @var = 99
  def self.value_of_var
    @var
  end
end
puts Test.value_of_var
```

produces:

```
99
```

The fact that self is set to the class during a class definition turns out to be a dramatically elegant decision, but to see why, we'll first need to have a look at singletons.

Singletons

Ruby lets you define methods that are specific to a particular object. These are called *singleton methods*. For example, let's start with a simple string object:

```
animal = "cat"
puts animal.upcase
```

produces:

```
CAT
```

This results in the object structure shown in Figure 24.1 on the previous page. The animal variable points to an object containing (among other things) the value of the string ("cat") and a pointer to the object's class, String.

When we call animal.upcase, Ruby goes to the object referenced by the animal variable and then looks up the method upcase in the class object referenced from the animal object. Our animal is a string and so has the methods of class String available.

Now let's make it more interesting by defining a singleton method on the string referenced from animal:

```
def animal.speak
  puts "The #{self} says miaow"
end
animal.speak
puts animal.upcase
```

produces:

```
The cat says miaow
CAT
```

We've already seen how the call to animal.speak works when we looked at how methods are invoked. Ruby sets self to the string object "cat" referenced by animal and then looks for a method speak in that object's class. Surprisingly, it finds it. It's initially surprising because the class of "cat" is String, and String doesn't have a speak method. So, does Ruby have some kind of special-case magic for these methods that are defined on individual objects?

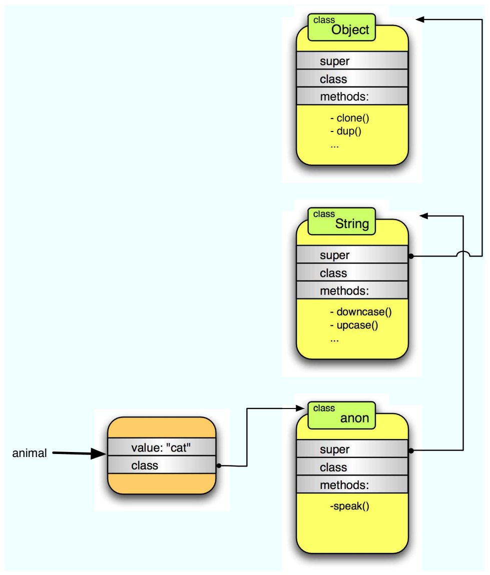
Thankfully, the answer is "no." Ruby's object model is remarkably consistent. When we defined the singleton method for the "cat" object, Ruby created a new anonymous class and defined the speak method in that class. This anonymous class is sometimes called a *singleton class* and other times an *eigenclass*. I prefer the former, because it ties in to the idea of singleton methods.

Ruby makes this singleton class the class of the "cat" object and makes String (which was the original class of "cat") the superclass of the singleton class. This is shown in Figure 24.2 on the following page.

Now let's follow the call to animal.speak. Ruby goes to the object referenced by animal and then looks in its class for the method speak. The class of the animal object is the newly created singleton class, and it contains the method we need.

What happens if we instead call animal.upcase? The processing starts the same way: Ruby looks for the method upcase in the singleton class but fails to find it there. It then follows the normal processing rules and starts looking up the chain of superclasses. The superclass of the singleton is String, and Ruby finds the upcase method there. Notice that there is no special-case processing here—Ruby method calls always work the same way.

Figure 24.2. Object Model for a Singleton Class



Singletons and Classes

Earlier, we said that inside a class definition, self is set to the class object being defined. It turns out that this is the basis for one of the more elegant aspects of Ruby's object model.

Recall that we can define class methods in Ruby using either of the forms def self.xxx or def ClassName.xxx:

Download **samples/classes_6.rb**

```
class Dave
  def self.class_method_one
    puts "Class method one"
  end
  def Dave.class_method_two
    puts "Class method two"
  end
end
Dave.class_method_one
Dave.class_method_two
```

produces:

```
Class method one
Class method two
```

Now we know why the two forms are identical: inside the class definition, self is set to Dave.

But now that we've looked at singleton methods, we also know that, in reality, there are no such thing as class methods in Ruby. Both of the previous definitions define singleton methods on the class object. As with all other singleton methods, we can then call them via the object (in this case, the class Dave).

Before we created the two singleton methods in class Dave, the class pointer in the class object pointed to class Class. (That's a confusing sentence. Another way of saying it is "Dave is a class, so the class of Dave is class Class," but that's pretty confusing, too.) The situation looks like Figure 24.3 on the next page.

The object diagram for class Dave after the methods are defined is shown in Figure 24.4 on page 392. Do you see how the singleton class is created, just as it was for the *animal* example? The class is inserted as the class of Dave, and the original class of Dave is made this new class's parent.

We can now tie together the two uses of self, the current object. We talked about how instance variables are looked up in self, and we talked about how singleton methods defined on self become class methods. Let's use these facts to access instance variables for class objects:

Download **samples/classes_7.rb**

```
class Test
  @var = 99
  def self.var
    @var
  end
  def self.var=(value)
    @var = value
  end
end
puts "Original value = #{Test.var}"
Test.var = "cat"
puts "New value = #{Test.var}"
```

produces:

```
Original value = 99
New value = cat
```

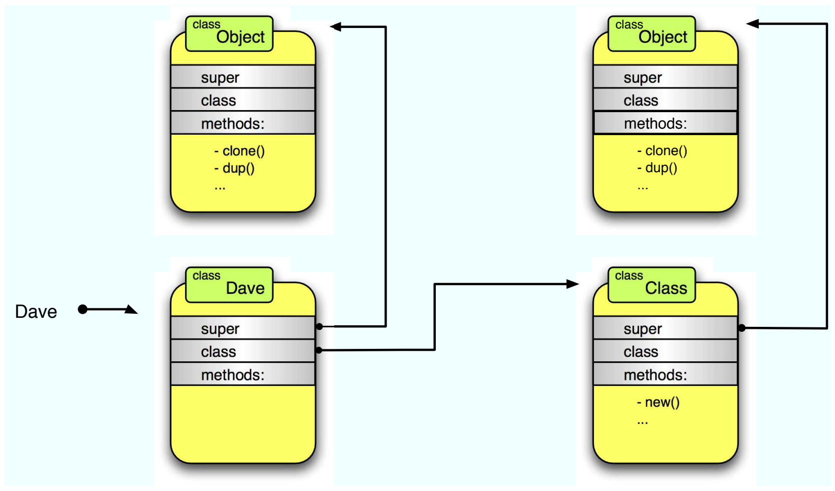
Newcomers to Ruby commonly make the mistake of setting instance variables inline in the class definition (as we did with @var in the previous code) and then attempting to access these variables from instance methods. As the code illustrates, this won't work, because instance variables defined in the class body are associated with the class object, not with instances of the class.

Another Way to Access the Singleton Class

We've seen how you can create methods in an object's singleton class by adding the object reference to the method definition using something like def animal.speak.

You can do the same using Ruby's class << an_object notation:

Figure 24.3. Basic Class Definition



Download **samples/classes_8.rb**

```
animal = "dog"
class << animal
  def speak
    puts "The #{self} says WOOF!"
  end
end
animal.speak
```

produces:

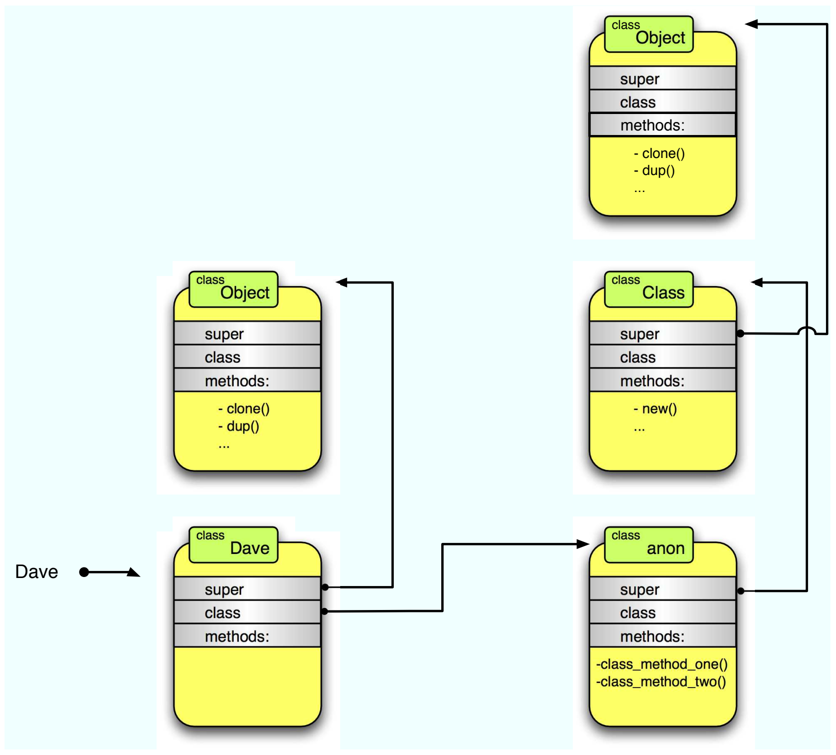
```
The dog says WOOF!
```

Inside this kind of class definition, self is set to the singleton class for the given object (animal in this case). Because class definitions return the value of the last statement executed in the class body, we can use this fact to get the singleton class object:

Download **samples/classes_9.rb**

```
animal = "dog"
def animal.speak
  puts "The #{self} says WOOF!"
end
singleton = class << animal
  def lie
    puts "The #{self} lies down"
  end
  self     # << return singleton class object
end
```

Figure 24.4. Class with So-Called Class Methods



```
animal.speak
animal.lie
puts "Singleton class object is #{singleton}"
puts "It defines methods #{singleton.instance_methods - 'cat'.methods}"
```

produces:

```
The dog says WOOF!
The dog lies down
Singleton class object is #<Class:#<String:0x0a36d8>>
It defines methods [:speak, :lie]
```

Note the notation that Ruby uses to denote a singleton class: #<Class:#<String:...> >.

Ruby goes to some trouble to stop you from using singleton classes outside the context of their original object. For example, you can't create a new instance of a singleton class:

Download **samples/classes_10.rb**

```
singleton = class << "cat"; self; end
singleton.new
```

produces:

```
prog.rb:2:in `new': can't create instance of singleton class (TypeError)
from /tmp/prog.rb:2:in `<main>'
```

Let's tie together what we know about instance variables, self, and singleton classes. Back on page 390, we wrote class-level accessor methods to let us get and set the value of an instance variable defined in a class object. But Ruby already has attr_accessor, which defines getter and setter methods. Normally, though, these are defined as instance methods and hence will access values stored in instances of a class. To make them work with class-level instance variables, we have to invoke attr_accessor in the singleton class:

Download **samples/classes_11.rb**

```
class Test
  @var = 99
  class << self
    attr_accessor :var
  end
end
puts "Original value = #{Test.var}"
Test.var = "cat"
puts "New value = #{Test.var}"
```

produces:

```
Original value = 99
New value = cat
```

Inheritance and Visibility

There's a wrinkle to when it comes to method definition and class inheritance, but it's fairly obscure. Within a class definition, you can change the visibility of a method in an ancestor class.

For example, you can do something like this:

Download **samples/classes_12.rb**

```
class Base
  def a_method
    puts "Got here"
  end
  private :a_method
end
class Derived1 < Base
  public :a_method
end
class Derived2 < Base
end
```

In this example, you would be able to invoke a_method in instances of class Derived1 but not via instances of Base or Derived2.

So, how does Ruby pull off this feat of having one method with two different visibilities? Simply put, it cheats.

If a subclass changes the visibility of a method in a parent, Ruby effectively inserts a hidden proxy method in the subclass that invokes the original method using super. It then sets the visibility of that proxy to whatever you requested. This means that the following code:

```
class Derived1 < Base
  public :a_method
end
```

is effectively the same as this:

```
class Derived1 < Base
  def a_method(*)
    super
  end
  public :a_method
end
```

The call to super can access the parent's method regardless of its visibility, so the rewrite allows the subclass to override its parent's visibility rules. Pretty scary, eh?

Modules and Mixins

You know that when you include a module into a Ruby class, the instance methods in that module become available as instance methods of the class.

Download **samples/classes_15.rb**

```
module Logger
  def log(msg)
    STDERR.puts Time.now.strftime("%H:%M:%S: ") + "#{self} (#{msg})"
  end
end
class Song
  include Logger
end
class Album
  include Logger
end
s = Song.new
s.log("created")
```

produces:

```
13:26:13: #<Song:0x0a323c> (created)
```

Ruby implements include very simply: the module that you include is effectively added as a superclass of the class being defined. It's as if the module was the parent of the class that it is mixed in to. And that would be the end of the description except for one small wrinkle. Because the module is injected into the chain of superclasses, it must itself hold a link to the original parent class. If it didn't, there'd be no way of traversing the superclass chain to look up methods. However, you can mix the same module into many different classes, and those classes could potentially have totally different superclass chains. If there were just one module object that we mixed in to all these classes, there'd be no way of keeping track of the different superclasses for each.

To get around this, Ruby uses a clever trick. When you include a module in class Example, Ruby constructs a new class object, makes it the superclass of Example, and then sets the superclass of the new class to be the original superclass of Example. It then references the module from this new class object in such a way that when you look a method up in this class, it actually looks it up in the module, as shown in Figure 24.5 on the following page.

A nice side effect of this arrangement is that if you change a module after including it in a class, those changes are reflected in the class (and the class's objects). In this way, modules behave just like classes.

Download **samples/classes_16.rb**

```
module Mod
  def greeting
    "Hello"
  end
end
class Example
  include Mod
end
ex = Example.new
puts "Before change, greeting is #{ex.greeting}"
module Mod
  def greeting
    "Hi"
  end
end
puts "After change, greeting is #{ex.greeting}"
```

produces:

```
Before change, greeting is Hello
After change, greeting is Hi
```

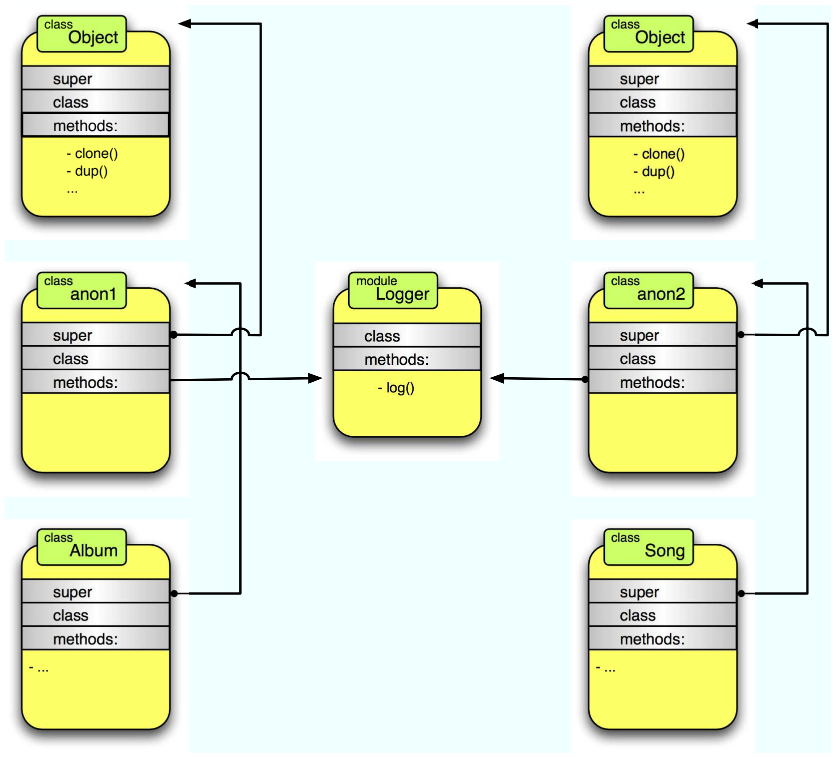
If a module itself includes other modules, a chain of proxy classes will be added to any class that includes that module, one proxy for each module that is directly or indirectly included.

Finally, Ruby will include a module only once in an inheritance chain—including a module that is already included by one of your superclasses is a no-op.

extend

The include method effectively adds a module as a superclass of self. It is used inside a class definition to make the instance methods in the module available to instances of the class.

Figure 24.5. How Modules Are Included



However, it is sometimes useful to add the instance methods to a particular object. You do this using Object#extend. For example:

Download **samples/classes_17.rb**

```
module Humor
  def tickle
    "#{self} says hee, hee!"
  end
end
obj = "Grouchy"
obj.extend Humor
puts obj.tickle
```

produces:

```
Grouchy says hee, hee!
```

Stop for a second to think about how this might be implemented....

When Ruby executes obj.tickle in this code example, it does the usual trick of looking in the class of obj for a method called tickle. For extend to work, it has to add the instance methods in the Humor module into the superclass chain for the class of obj. So, just as with singleton method definitions, Ruby creates a singleton class for obj and then includes the module Humor in that class. In fact, just to prove that this is all that happens, here's the C implementation of extend in the current Ruby 1.9 interpreter:

```
void
rb_extend_object(VALUE obj, VALUE module)
{
    rb_include_module(rb_singleton_class(obj), module);
}
```

There is an interesting trick with extend. If you use it within a class definition, the module's methods become class methods. This is because calling extend is equivalent to self.extend, so the methods are added to self, which in a class definition is the class itself.

Here's an example of adding a module's methods at the class level:

Download **samples/classes_19.rb**

```
module Humor
  def tickle
    "#{self} says hee, hee!"
  end
end
class Grouchy
  extend  Humor
end
puts Grouchy.tickle
```

produces:

```
Grouchy says hee, hee!
```

Later, on page 400, we'll see how to use extend to add macro-style methods to a class.

Metaprogramming Class-Level Macros

If you've used Ruby for any time at all, the chances are good that you've used attr_accessor, the method that defines reader and writer methods for instance variables:

```
class Song
  attr_accessor :duration
end
```

If you've written a Ruby on Rails application, you've probably used has_many:

```
class Album < ActiveRecord::Base
  has_many :tracks
end
```

These are both examples of class-level methods that generate code behind the scenes. Because of the way they expand into something bigger, folks sometimes call these kinds of methods *macros*.

Let's create a trivial example and then build it up into something realistic. We'll start by implementing a simple method that adds logging capabilities to instances of a class. We previously did this using a module—this time we'll do it using a class-level method. Here's the first iteration:

Download **samples/classes_22.rb**

```
class Example
  def self.add_logging
    def log(msg)
      STDERR.puts Time.now.strftime("%H:%M:%S: ") + "#{self} (#{msg})"
    end
  end
  add_logging
end
ex = Example.new
ex.log("hello")
```

produces:

```
13:26:13: #<Example:0x0a39a8> (hello)
```

Clearly, this is a silly piece of code. But bear with me—it'll get better. And we can still learn some stuff from it. First, notice that add_logging is a class-method—it is defined in the class object's singleton class. That means that we can call it later in the class definition without an explicit receiver, because self is set to the class object inside a class definition.

Then, notice that the add_logging method contains a nested method definition. This inner definition will get executed only when we call the add_logging method. The result is that log will be defined as an instance method of class Example.

Let's take one more step. We can define the add_logging method in one class and then use it in a subclass. This works because the singleton class hierarchy parallels the regular class hierarchy. As a result, class methods in a parent class are also available in the child class:

Download **samples/classes_23.rb**

```
class Logger
  def self.add_logging
    def log(msg)
      STDERR.puts Time.now.strftime("%H:%M:%S: ") + "#{self} (#{msg})"
    end
  end
end
class Example < Logger
  add_logging
end
ex = Example.new
ex.log("hello")
```

produces:

```
13:26:13: #<Example:0x0a34d0> (hello)
```

Think back to the two examples at the start of this section. Both work this way. attr_accessor is a class method defined in class Module and so is available in all module and class definitions. has_many is a class method defined in the Base class within the Rails ActiveRecord module and so is available to all classes that subclass ActiveRecord::Base.

This example is still not particularly compelling; it would still be easier to add the log method directly as an instance method of our Logger class. But what happens if we want to construct a different version of the log method for each class that uses it? For example, let's add the capability to add a short class-specific identifying string to the start of each log message. We want to be able to say something like this:

Download **samples/classes_24.rb**

```
class Song < Logger
  add_logging "Song"
end
class Album < Logger
  add_logging "CD"
end
```

To do this, let's define the log method on the fly. We can no longer use a straightforward def ... end-style definition. Instead, we'll use define_method, one of the cornerstones of metaprogramming. define_method takes the name of a method and a block, defining a method with the given name and with the block as the method body. Any arguments in the block definition become parameters to the method being defined.

Download **samples/classes_25.rb**

```
class Logger
  def self.add_logging(id_string)
    define_method(:log) do |msg|
      now = Time.now.strftime("%H:%M:%S")
      STDERR.puts "#{now}-#{id_string}: #{self} (#{msg})"
    end
  end
end
class Song < Logger
  add_logging "Tune"
end
class Album < Logger
  add_logging "CD"
end
song = Song.new
song.log("rock on")
```

produces:

```
13:26:13-Tune: #<Song:0x0a20e4> (rock on)
```

There's an important subtlety in this code. Notice that the body of the log method contains this line:

```
STDERR.puts "#{now}-#{id_string}: #{self} (#{msg})"
```

The value now is a local variable, and msg is the parameter to the block. But id_string is the parameter to the enclosing add_logging method. It's accessible inside the block because block definitions create closures, allowing the context in which the block is defined to be carried forward and used when the block is used. In this case, we're taking a value from a class-level method and using it in an instance method we're defining. This is a common pattern when creating these kinds of class-level macros.

As well as passing parameters from the class method into the body of the method being defined, we can also use the parameter to determine the name of the method or methods to create. Here's an example that creates a new kind of attr_accessor that logs all assignments to a given instance variable:

Download **samples/classes_27.rb**

```
class AttrLogger
  def self.attr_logger(name)
    attr_reader name
    define_method("#{name}=") do |val|
      puts "Assigning #{val.inspect} to #{name}"
      instance_variable_set("@#{name}", val)
    end
  end
end
class Example < AttrLogger
  attr_logger :value
end
ex = Example.new
ex.value = 123
puts "Value is #{ex.value}"
ex.value = "cat"
puts "Value is now #{ex.value}"
```

produces:

```
Assigning 123 to value
Value is 123
Assigning "cat" to value
Value is now cat
```

Again, we use the fact that the block defining the method body is a closure, accessing the name of the attribute in the log message string. Notice we also make use of the fact that attr_reader is simply a class method—we can call it inside our class method to define the reader method for our attribute. Note another common bit of metaprogramming—we use instance_variable_set to set the value of an instance variable (duh). There's a corresponding _get method that fetches the value of a named instance variable.

Class Macros and Modules

Sometimes it is perfectly acceptable to define class macros in one class and then use these macro methods in subclasses of this class. Other times, though, it isn't appropriate to use subclassing, either because we already have to subclass some other class or because our design aesthetic rebels against making something like a song a subclass of a logger.

In these cases, you can use a module to hold your metaprogramming implementation. As we've seen, using extend inside a class definition will add the methods in a module as class methods to the class being defined:

Download **samples/classes_28.rb**

```
module AttrLogger
  def attr_logger(name)
    attr_reader name
    define_method("#{name}=") do |val|
      puts "Assigning #{val.inspect} to #{name}"
      instance_variable_set("@#{name}", val)
    end
  end
end
class Example
  extend AttrLogger
  attr_logger :value
end
ex = Example.new
ex.value = 123
puts "Value is #{ex.value}"
ex.value = "cat"
puts "Value is now #{ex.value}"
```

produces:

```
Assigning 123 to value
Value is 123
Assigning "cat" to value
Value is now cat
```

Things get a little trickier if you want to add both class methods and instance methods into the class being defined. Here's one technique, used extensively in the implementation of the Rails framework. It makes use of a Ruby hook method, included, which is called automatically by Ruby when you include a module into a class. It is passed the class object of the class being defined.

Download **samples/classes_29.rb**

```
module GeneralLogger
  # Instance method to be added to any class that includes us
  def log(msg)
    puts Time.now.strftime("%H:%M: ") + msg
  end
  # module containing class methods to be added
  module ClassMethods
    def attr_logger(name)
      attr_reader name
      define_method("#{name}=") do |val|
        log "Assigning #{val.inspect} to #{name}"
        instance_variable_set("@#{name}", val)
      end
    end
  end
  # extend host class with class methods when we're included
  def self.included(host_class)
    host_class.extend(ClassMethods)
  end
end
class Example
  include GeneralLogger
  attr_logger :value
end
ex = Example.new
ex.log("New example created")
ex.value = 123
puts "Value is #{ex.value}"
ex.value = "cat"
puts "Value is #{ex.value}"
```

produces:

```
13:26: New example created
13:26: Assigning 123 to value
Value is 123
13:26: Assigning "cat" to value
Value is cat
```

Notice how the included callback is used to extend the host class with the methods defined in the inner module ClassMethods.

Now, as an exercise, try executing the previous example in your head. For each line of code, work out the value of self. Master this, and you've pretty much mastered this style of metaprogramming in Ruby.

Two Other Forms of Class Definition

Just in case you thought we'd exhausted the ways of defining Ruby classes, let's look at two other options.

Subclassing Expressions

The first form is really nothing new—it's simply a generalization of the regular class definition syntax. You know that you can write this:

```
class Parent
  ...
end
class Child < Parent
  ...
end
```

What you might not know is that the thing to the right of the < needn't be just a class name; it can be any expression that returns a class object. In this code example, we have the constant Parent. A constant is a simple form of expression, and in this case the constant Parent holds the class object of the first class we defined.

Ruby comes with a class called Struct, which allows you to define classes that contain just data attributes. For example, you could write this:

Download **samples/classes_31.rb**

```
Person = Struct.new(:name, :address, :likes)
dave = Person.new('Dave', 'TX')
dave.likes = "Programming Languages"
puts dave
```

produces:

```
#<struct Person name="Dave", address="TX", likes="Programming Languages">
```

The return value from Struct.new(...) is a class object. By assigning it to the constant Person, we can thereafter use Person as if it were any other class.

But say we wanted to change the to_s method of our structure.

We could do it by opening up the class and writing the method:

Download **samples/classes_32.rb**

```
Person = Struct.new(:name, :address, :likes)
class Person
  def to_s
    "#{self.name} lives in #{self.address} and likes #{self.likes}"
  end
end
```

However, we can do this more elegantly (although at the cost of an additional class object) by writing this:

Download **samples/classes_33.rb**

```
class Person < Struct.new(:name, :address, :likes)
  def to_s
    "#{self.name} lives in #{self.address} and likes #{self.likes}"
  end
end
dave = Person.new('Dave', 'Texas')
dave.likes = "Programming Languages"
puts dave
```

produces:

```
Dave lives in Texas and likes Programming Languages
```

Creating Singleton Classes

Let's look at some Ruby code:

```
class Example
end
ex = Example.new
```

When we call Example.new, we're invoking the method new on the class object Example. This is just a regular method call—Ruby looks for the method new in the class of the object (and the class of Example is Class) and invokes it. It turns out that we can also invoke Class#new directly:

```
some_class = Class.new
puts some_class.class
```

produces:

```
Class
```

If you pass Class.new a block, that block is used as the body of the class:

Download **samples/classes_36.rb**

```
some_class = Class.new do
  def self.class_method
    puts "In class method"
  end
  def instance_method
    puts "In instance method"
  end
end
some_class.class_method
obj = some_class.new
obj.instance_method
```

produces:

```
In class method
In instance method
```

By default, these classes will be direct descendents of Object. You can give them a different parent by passing the parent's class as a parameter:

Download **samples/classes_37.rb**

```
some_class = Class.new(String) do
  def vowel_movement
    tr 'aeiou', '*'
  end
end
obj = some_class.new("now is the time")
puts obj.vowel_movement
```

produces:

```
n*w *s th* t*m*
```

How Classes Get Their Names

You may have noticed that the classes created by Class.new have no name. However, all is not lost. If you assign the class object for a class with no name to a constant, Ruby will automatically name the class after the constant:

```
some_class = Class.new
obj = some_class.new
puts "Initial name is #{some_class.name}"
SomeClass = some_class
puts "Then the name is #{some_class.name}"
puts "also works via the object: #{obj.class.name}"
```

produces:

```
Initial name is
Then the name is SomeClass
also works via the object: SomeClass
```

We can use these dynamically constructed classes to extend Ruby in interesting ways. For example, here's a simple reimplementation of the Ruby Struct class:

Download **samples/classes_39.rb**

```
def MyStruct(*keys)
  Class.new do
    attr_accessor *keys
    def initialize(hash)
      hash.each do |key, value|
        instance_variable_set("@#{key}", value)
      end
    end
  end
end
Person = MyStruct :name, :address, :likes
dave = Person.new(name: "dave", address: "TX", likes: "Stilton")
chad = Person.new(name: "chad", likes: "Jazz")
chad.address = "CO"
puts "Dave's name is #{dave.name}"
puts "Chad lives in #{chad.address}"
```

produces:

```
Dave's name is dave
Chad lives in CO
```

instance_eval and class_eval

The methods Object#instance_eval, Object#class_eval, and Object#module_eval let you set self to be some arbitrary object, evaluate the code in a block with, and then reset self:

```
"cat".instance_eval do
  puts "Upper case = #{upcase}"
  puts "Length is #{self.length}"
end
```

produces:

```
Upper case = CAT
Length is 3
```

Both forms also take a string (but see the sidebar on the following page for some notes on the dangers of evaluating strings):

```
"cat".instance_eval('puts "Upper=#{upcase}, length=#{self.length}"')
```

produces:

```
Upper=CAT, length=3
```

class_eval and instance_eval both set self for the duration of the block. However, they differ in the way they set up the environment for method definition. class_eval sets things up as if you were in the body of a class definition, so method definitions will define instance methods:

```
class MyClass
end
MyClass.class_eval do
  def instance_method
    puts "In an instance method"
  end
end
obj = MyClass.new
obj.instance_method
```

produces:

```
In an instance method
```

In contrast, instance_eval acts as if you were working inside the singleton class of self. Therefore, any methods you define will become class methods.

```
class MyClass
end
MyClass.instance_eval do
  def class_method
    puts "In a class method"
  end
end
MyClass.class_method
```

produces:

```
In a class method
```

eval Is Soo Last Year

You may have noticed that we've been doing a fair amount of metaprogramming—accessing instance variables, defining methods, and creating classes—and we haven't yet used eval. This is deliberate. In the old days of Ruby, the language lacked many of these metaprogramming facilities, and eval was the only way of achieving these effects. But eval comes with a couple of downsides.

First, it is slow—calling eval effectively compiles the code in the string before executing it. But, even worse, eval can be dangerous. If there's any chance that external data—stuff that comes from outside your application—can wind up inside the parameter to eval, then you have a security hole, because that external data may end up containing arbitrary code that your application will blindly execute.

eval is now considered a method of last resort.

It might be helpful to remember that, when defining methods, class_eval and instance_eval have precisely the wrong names: class_eval defines instance methods, and instance_eval defines class methods. Go figure.

1.9 Ruby 1.9 introduces variants of these methods. Object#instance_exec, Module#class_exec, and Module#module_exec behave identically to their _eval counterparts but take only a block (that is, they do not take a string). Any arguments given to the methods are passed in as block parameters. This is an important feature. Previously it was impossible to pass a local or instance variable into a block given to one of the _eval methods—because self is changed by the call, these variables go out of scope. With the _exec form, you can now pass them in:

Download **samples/classes_44.rb**

```
animal = "cat"
"dog".instance_exec(animal) do |other|
  puts "#{other} and #{self}"
end
```

produces:

```
cat and dog
```

instance_eval and Constants

1.9 Ruby 1.9 has changed the way Ruby looks up constants when executing a block using instance_eval and class_eval. Previously, constants were looked up in the lexical scope in which there were referenced. In Ruby 1.9, they are now looked up in the scope in which instance_eval is called. This (artificial) example shows the output produced by Ruby 1.9:

Download samples/classes_45.rb

```
module One
  CONST = "Defined in One"
  def self.eval_block(&block)
    instance_eval(&block)
  end
end
module Two
  CONST = "Defined in Two"
  def self.call_eval_block
    One.eval_block do
      puts CONST
    end
  end
end
Two.call_eval_block
```

produces:

```
Defined in One
```

In Ruby 1.8, this same code would print Defined in Two.

instance_eval and Domain-Specific Languages

It turns out that instance_eval has a pivotal role to play in a certain type of domain-specific language (DSL). For example, we might be writing a simple DSL for turtle graphics.[2] To draw a set of three 5x5 squares, we might write this:[3]

```
3.times do
  forward(8)
  pen_down
  4.times do
    forward(4)
    left
  end
  pen_up
end
```

Clearly, pen_down, forward, left, and pen_up can be implemented as Ruby methods. However, to call them without a receiver like this, either we have to be within a class that defines them (or is a child of such a class) or we have to make the methods global. instance_eval to the rescue. We can define a class Turtle that defines the various methods we need as instance methods. We'll also define a walk method, which will execute our turtle DSL, and a draw method to draw the resulting picture:

2. In turtle graphics systems, you imagine you have a turtle you can command to move forward n squares, turn left, and turn right. You can also make the turtle raise and lower a pen. If the pen is lowered, a line will be drawn tracing the turtle's subsequent movements. Very few of these turtles exist in the wild, so we tend to simulate them inside computers.

3. Yes, the forward(4) is correct in this code. The initial point is always drawn.

```
class Turtle
  def left; ... end
  def right; ... end
  def forward(n); ... end
  def pen_up; .. end
  def pen_down; ... end
  def walk(...); end
  def draw; ... end
end
```

If we implement walk correctly, we can then write this:

```
turtle = Turtle.new
turtle.walk do
  3.times do
    forward(8)
    pen_down
    4.times do
      forward(4)
      left
    end
    pen_up
  end
end
turtle.draw
```

So, what is the correct implementation of walk? Well, we clearly have to use instance_eval, because we want the DSL commands in the block to call the methods in the turtle object. We also have to arrange to pass the block given to the walk method to be evaluated by that instance_eval call. Our implementation looks like this:

```
def walk(&block)
  instance_eval(&block)
end
```

Notice how we captured the block into a variable and then expanded that variable back into a block in the call to instance_eval.

A complete listing of the turtle program starts on page 418.

Is this a good use of instance_eval? It depends on the circumstances. The benefit is that the code inside the block looks simple—you don't have to make the receiver explicit:

```
4.times do
  turtle.forward(4)
  turtle.left
end
```

There's a drawback, though. Inside the block, scope isn't what you think it is, so this code wouldn't work:

```
@size = 4
turtle.walk do
  4.times do
    turtle.forward(@size)
    turtle.left
  end
end
```

Instance variables are looked up in self, and self in the block isn't the same as self in the code that sets the instance variable @size. Because of this, most people are moving away from this style of CFinstance_evaled block.

Hook Methods

In the section starting on page 400, we defined a method called included in our GeneralLogger module. When this module was included in a class, Ruby automatically invoked this included method, allowing our module to add class methods to the host class.

included is an example of a *hook method* (sometimes called a *callback*). A hook method is a method that you write but that Ruby calls from within the interpreter when some particular event occurs. The interpreter looks for these methods by name—if you define a method in the right context with an appropriate name, Ruby will call it when the corresponding event happens.

The methods that can be invoked from within the interpreter are shown in Table 24.1 on the next page. We won't discuss all of them in this chapter—instead, we'll show just a few examples of use. The reference section of this book describes the individual methods, and the *Duck Typing* chapter on page 370 discusses the coercion methods in more detail.

The `inherited` Hook

If a class defines a class method called inherited, Ruby will call it whenever that class is subclassed (that is, whenever any class inherits from the original).

This hook is often used in situations where a base class needs to keep track of its children. For example, an online store might offer a variety of shipping options. Each might be represented by a separate class, and each of these classes could be a subclass of a single Shipping class. This parent class could keep track of all the various shipping options by recording every class that subclasses it. When it comes time to display the shipping options to the user, the application could call the base class, asking it for a list of its children:

Download **samples/classes_52.rb**

```
class Shipping     # Base class
  @children = []   # this variable is in the class, not instances
  def self.inherited(child)
    @children << child
  end
  def self.shipping_options(weight, international)
    @children.select {|child| child.can_ship(weight, international)}
  end
end
```

Table 24.1. Ruby Hook Methods

Method-related hooks	method_added, method_missing, method_removed, method_undefined, singleton_method_added, singleton_method_removed, singleton_method_undefined
Class and module-related hooks	append_features, const_missing, extend_object, extended, included, inherited, initialize_copy
Object marshaling hooks	marshal_dump, marshal_load
Coercion hooks	coerce, induced_from, to_*xxx*

```
class MediaMail < Shipping
  def self.can_ship(weight, international)
    !international
  end
end
class FlatRatePriorityEnvelope < Shipping
  def self.can_ship(weight, international)
    weight < 64 && !international
  end
end
class InternationalFlatRateBox < Shipping
  def self.can_ship(weight, international)
    weight < 9*16 && international
  end
end
puts "Shipping 16oz domestic"
puts Shipping.shipping_options(16, false)
puts "\nShipping 90oz domestic"
puts Shipping.shipping_options(90, false)
puts "\nShipping 16oz international"
puts Shipping.shipping_options(16, true)
```

produces:

```
Shipping 16oz domestic
MediaMail
FlatRatePriorityEnvelope

Shipping 90oz domestic
MediaMail

Shipping 16oz international
InternationalFlatRateBox
```

Command interpreters are another common user of this pattern: the base class keeps a track of available commands, each of which is implemented in a subclass.

The method_missing Hook

Earlier, we saw how Ruby executes a method call by looking for the method, first in the object's class, then in its superclass, then in that class's superclass, and so on. If the method call has an explicit receiver, then private methods are skipped in this search. If the method is not found by the time we run out of superclasses (because BasicObject has no superclass), then Ruby tries to invoke the hook method method_missing on the original object. Again, the same process is followed—Ruby first looks in the object's class, then in its superclass, and so on. However, Ruby predefines its own version of method_missing in class BasicObject, so typically the search stops there. The built-in method_missing basically raises an exception (either a NoMethodError or a NameError depending on the circumstances).

The key here is that method_missing is simply a Ruby method. We can override it in our own classes to handle calls to otherwise undefined methods in an application-specific way.

method_missing has a simple signature, but many people get it wrong:

```
def method_missing(name, *args, &block)  # ...
```

The name argument receives the name of the method that couldn't be found. It is passed as a symbol. The args argument is an array of the arguments that were passed in the original call. And the oft-forgotten block argument will receive any block passed to the original method.

Download **samples/classes_54.rb**

```
def method_missing(name, *args, &block)
  puts "Called #{name} with #{args.inspect} and #{block}"
end
wibble
wobble 1, 2
wurble(3, 4) { stuff }
```

produces:

```
Called wibble with [] and
Called wobble with [1, 2] and
Called wurble with [3, 4] and #<Proc:0x0a3d68@/tmp/prog.rb:7>
```

Before we get too deep into the details, I'll offer a tip about etiquette. There are two main ways that people use method_missing. The first intercepts every use of an undefined method and handles it. The second is more subtle; it intercepts all calls but handles only some of them. In the latter case, it is important to forward on the call to a superclass if you decide not to handle it in your method_missing implementation:

Download **samples/classes_55.rb**

```
class MyClass < OtherClass
  def method_missing(name, *args, &block)
    if <some condition>
      # handle call
    else
      super   # otherwise pass it on
    end
  end
end
```

If you fail to pass on calls that you don't handle, your application will silently ignore calls to unknown methods in your class.

Let's show a couple of uses of method_missing.

method_missing to Simulate Accessors

The OpenStruct class is distributed with Ruby. It allows you to write objects with attributes that are created dynamically by assignment. (We describe it in more detail on page 787.) For example, you could write this:

Download **samples/classes_56.rb**

```
require 'ostruct'
obj = OpenStruct.new(name: "Dave")
obj.address = "Texas"
obj.likes   = "Programming"
puts "#{obj.name} lives in #{obj.address} and likes #{obj.likes}"
```

produces:

```
Dave lives in Texas and likes Programming
```

Let's use method_missing to write our own version of OpenStruct:

Download **samples/classes_57.rb**

```
class MyOpenStruct < BasicObject
  def initialize(initial_values = {})
    @values = initial_values
  end
  def _singleton_class
    class << self
      self
    end
  end
  def method_missing(name, *args, &block)
    if name[-1] == "="
      base_name = name[0..-2].intern
      _singleton_class.instance_exec(name) do |name|
        define_method(name) do |value|
          @values[base_name] = value
        end
      end
      @values[base_name] = args[0]
    else
      _singleton_class.instance_exec(name) do |name|
        define_method(name) do
          @values[name]
        end
      end
      @values[name]
    end
  end
end
```

```
obj = MyOpenStruct.new(name: "Dave")
obj.address = "Texas"
obj.likes   = "Programming"
puts "#{obj.name} lives in #{obj.address} and likes #{obj.likes}"
```

produces:

```
Dave lives in Texas and likes Programming
```

1.9 Notice how we base our class on BasicObject, a class introduced in Ruby 1.9. BasicObject is the root of Ruby's object hierarchy and contains only a minimal number of methods:

```
p BasicObject.instance_methods
```

produces:

```
[:==, :equal?, :!, :!=, :instance_eval, :instance_exec, :__send__]
```

This is good, because it means that our MyOpenStruct class will be able to have attributes such as display or class. If instead we'd based MyOpenStruct on class Object, then these names, along with 47 others, would have been predefined and hence wouldn't trigger method_missing.

Notice also another common pattern inside method_missing. The first time we reference or assign to an attribute of our object, we access or update the @values hash appropriately. But we also define the method that the caller was trying to access. This means that the next time this attribute is used, it will use the method and not invoke method_missing. This may or may no be worth the trouble, depending on the access patterns to your object.

Also notice how we had to jump through some hoops to define the method. We want to define the method only for the current object. This means we have to put the method into the object's singleton class. We can do that using instance_exec and define_method. But that means we have to use the class << self trick to get the object's singleton class. Through an interesting implementation subtlety, define_method will always define an instance method, independent of whether it is invoked via instance_exec or class_exec.

However, this code reveals a dark underbelly of using method_missing and BasicObject. Consider this:

```
obj = MyOpenStruct.new(name: "Dave")
obj.address = "Texas"
o1 = obj.dup
o1.name = "Mike"
o1.address = "Colorado"
```

produces:

```
prog.rb:5:in `<main>': undefined method `name=' for nil:NilClass (NoMethodError)
```

The dup method is not defined by BasicObject; it appears in class Object. So when we called dup, it was picked up by our method_missing handler, and we just returned nil (because we don't have yet have an attribute called dup). We could fix this so that it at least reports an error:

```
def method_missing(name, *args, &block)
  if name[-1] == "="
    # as before...
  else
    super unless @values.has_key? name
    # as before...
  end
end
```

This class now reports an error if we call dup (or any other method) on it. However, we still can't dup or clone it (or inspect, convert to a string, and so on). Although BasicObject seems like a natural fit for method_missing, you may find it to be more trouble than it's worth.

method_missing as a Filter

As the previous example showed, method_missing has some drawbacks if you use it to intercept all calls. It is probably better to use it to recognize certain patterns of call, passing on those it doesn't recognize to its parent class to handle.

An example of this is the dynamic finder facility in the Ruby on Rails ActiveRecord module. Active Record is the object-relational library in Rails—it allows you to access relational databases as if they were object stores. One particular feature allows you to find rows that match the criteria of having given values in certain columns. For example, if an Active Record class called Book was mapping a relational table called books and the books table included columns called title and author, you could write this:

```
pickaxe = Book.find_by_title("Programming Ruby")
daves_books = Book.find_all_by_author("Dave Thomas")
```

Active Record does not predefine all these potential finder methods. Instead, it uses our old friend method_missing. Inside that method, it looks for calls to undefined methods that match the pattern /^find_(all_)?by_(.*)/. [4] If the method being invoked does not match this pattern or if the field(s) in the method name don't correspond to columns in the database table, Active Record calls super so that a genuine method_missing report will be generated.

One Last Example

Let's bring together all of the metaprogramming topics we've discussed in a final example by writing a module that allows us to trace the execution of methods in any class that mixes the module in. This would let us write:

```
require 'code/trace_calls'
class Example
  def one(arg)
    puts "One called with #{arg}"
  end
end
```

4. It also looks for /^find_or_(initialize|create)_by_(.*)/.

```
ex1 = Example.new
ex1.one("Hello")     # no tracing from this call
class Example
  include TraceCalls
  def two(arg1, arg2)
    arg1 + arg2
  end
end
ex1.one("Goodbye")   # but we see tracing from these two
puts ex1.two(4, 5)
```

produces:

```
One called with Hello
==> calling one with ["Goodbye"]
One called with Goodbye
<== one returned nil
==> calling two with [4, 5]
<== two returned 9
9
```

We can see immediately that there's a subtlety here. When we mix the TraceCalls module into a class, it has to add tracing to any existing instance methods in that class. It also has to arrange to add tracing to any methods we subsequently add.

Let's start with the full listing of the TraceCalls module:

Download **samples/trace_calls.rb**

```
module TraceCalls
  def self.included(klass)
    klass.instance_methods(false).each do |existing_method|
      wrap(klass, existing_method)
    end
    def klass.method_added(method)  # note: nested definition
      unless @trace_calls_internal
        @trace_calls_internal = true
        TraceCalls.wrap(self, method)
        @trace_calls_internal = false
      end
    end
  end
  def self.wrap(klass, method)
    klass.instance_eval do
      method_object = instance_method(method)
      define_method(method) do |*args, &block|
        puts "==> calling #{method} with #{args.inspect}"
        result = method_object.bind(self).call(*args, &block)
        puts "<== #{method} returned #{result.inspect}"
        result
      end
    end
  end
end
```

When we include this module in a class, the included hook method gets invoked. It first uses the instance_methods reflection method to find all the existing instance methods in the host class (the false parameter limits the list to methods in the class itself, and not in its superclasses). For each existing method, the module calls a helper method, wrap, to add some tracing code to it. We'll talk about wrap shortly.

Next, the included method uses another hook, method_added. This is called by Ruby whenever a method is defined in the receiver. Note that we define this method in the class passed to the included method. This means that the method will be called when methods are added to this host class and not to the module. This is what allows us to include TraceCalls at the top of a class and then add methods to that class—all those method definitions will be handled by method_added.

Now look at the code inside the method_added method. We have to deal with a potential problem here. As you'll see when we look at the wrap method, we add tracing to a method by creating a new version of the method that calls the old. Inside method_added, we call the wrap function to add this tracing. But inside wrap, we'll define a new method to handle this wrapping, and that definition will invoke method_added again, and then we'd call wrap again, and so on, until the stack gets exhausted. To prevent this, we use an instance variable and do the wrapping only if we're not already doing it.

The wrap method takes a class object and the name of a method to wrap. It finds the original definition of that method (using instance_method) and saves it. It then redefines this method. This new method outputs some tracing and then calls the original, passing in the parameters and block from the wrapper.[5] Note how we call the method by binding the method object to the current instance and then invoking that bound method.

The key to understanding this code, and most metaprogramming code, is to follow the basic principles we worked out at the start of this chapter—how self changes as methods are called and classes are defined and how methods are called by looking for them in the class of the receiver. If you get stuck, do what I do and draw little boxes and arrows. I find it useful to stick with the convention I used in this chapter: class links go to the right, and superclass links go up. Given an object, a method call is then a question of finding the receiver object, going right once, and then following the superclass chain up as far as you need to go.

Top-Level Execution Environment

Finally, there's one small detail we have to cover to complete the metaprogramming environment. Many times in this book we've claimed that everything in Ruby is an object. However, we've used one thing time and time again that appears to contradict this—the top-level Ruby execution environment:

```
puts "Hello, World"
```

5. The ability of a block to take a block parameter was added in Ruby 1.9.

Not an object in sight. We may as well be writing some variant of Fortran or BASIC. But dig deeper, and you'll come across objects and classes lurking in even the simplest code.

We know that the literal "Hello, World" generates a Ruby String, so that's one object. We also know that the bare method call to puts is effectively the same as self.puts. But what is self?

```
self.class   # =>   Object
```

At the top level, we're executing code in the context of some predefined object. When we define methods, we're actually creating (private) instance methods for class Object. This is fairly subtle; as they are in class Object, these methods are available everywhere. And because we're in the context of Object, we can use all of Object's methods (including those mixed in from Kernel) in function form. This explains why we can call Kernel methods such as puts at the top level (and indeed throughout Ruby); it's because these methods are part of every object. Top-level instance variables also belong to this top-level object.

Metaprogramming is one of Ruby's sharpest tools. Don't be afraid to use it to raise up the level at which you program. But, at the same time, use it only when necessary—overly metaprogrammed applications can become pretty obscure pretty quickly.

The Turtle Graphics Program

Download **samples/classes_66.rb**

```
class Turtle
  # directions: 0 = E, 1 = S, 2 = W, 3 = N
  # axis: 0 = x, 1 = y
  def initialize
    @board = Hash.new(" ")
    @x = @y = 0
    @direction = 0
    pen_up
  end
  def pen_up
    @pen_down = false
  end
  def pen_down
    @pen_down = true
    mark_current_location
  end
  def forward(n=1)
    n.times { move }
  end
  def left
    @direction -= 1
    @direction = 3 if @direction < 0
  end
  def right
    @direction += 1
    @direction = 0 if @direction > 3
  end
```

```
    def walk(&block)
      instance_eval(&block)
    end
    def draw
      min_x, max_x = @board.keys.map{|x,y| x}.minmax
      min_y, max_y = @board.keys.map{|x,y| y}.minmax
      min_y.upto(max_y) do |y|
        min_x.upto(max_x) do |x|
          print @board[[x,y]]
        end
        puts
      end
    end
    private
    def move
      increment = @direction > 1 ? -1 : 1
      if @direction.even?
        @x += increment
      else
        @y += increment
      end
      mark_current_location
    end
    def mark_current_location
      @board[[@x,@y]] = "#" if @pen_down
    end
  end
  turtle = Turtle.new
  turtle.walk do
    3.times do
      forward(8)
      pen_down
      4.times do
        forward(4)
        left
      end
      pen_up
    end
  end
  turtle.draw
```

produces:

```
#####   #####   #####
#   #   #   #   #   #
#   #   #   #   #   #
#   #   #   #   #   #
#####   #####   #####
```