Table of Contents

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Chapter 5. OOP and Dynamicity in Ruby

IN THIS CHAPTER

> *Just as the introduction of the irrational numbers… is a convenient myth [which] simplifies the laws of arithmetic… so physical objects are postulated entities which round out and simplify our account of the flux of existence…. The conceptional scheme of physical objects is [likewise] a convenient myth, simpler than the literal truth and yet containing that literal truth as a scattered part.*
> —Willard Van Orman Quine

This is an unusual chapter. Whereas most of the chapters in this book deal with a specific problem domain, this one doesn't. If the problem space is viewed as stretching out on one axis of a graph, this chapter extends out on the other axis, encompassing a slice of each of the problem domains. This is because object-oriented programming and dynamicity aren't problem domains themselves, but are paradigms that can be applied to any problem whether it be system administration, low-level networking, or Web development.

For this reason, much of this chapter's information should already be familiar to a programmer who knows Ruby. In fact, the rest of the book wouldn't make sense without some of the fundamental knowledge here. Any Ruby programmer knows how to create a subclass, for instance.

This raises the question of what to include and what to exclude. Does every Ruby programmer know about the `extend` method? What about the `instance_eval` method? What is obvious to one person might be big news to another.

We have decided to err on the side of completeness. We include in this chapter some of the more esoteric tasks you might want to do with dynamic OOP in Ruby, but we also include the more routine tasks in case anyone is unfamiliar with them. We go right down to the simplest level because people don't agree on where the middle level ends. And we have tried to offer a little extra information even on the most basic of topics to justify their inclusion here. On the other hand, topics that are fully covered elsewhere in the book are omitted here.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

We'll also make two other comments. First of all, there is nothing magical about dynamic OOP. Ruby's object orientation and its dynamic nature do interact with each other, but they aren't inherently interrelated; we put them in a single chapter largely for convenience. Second, some language features might be mentioned here that aren't strictly related to either topic. Consider this to be cheating, if you will. We wanted to put them *somewhere*.

Everyday OOP Tasks

> *Of his quick objects hath the mind no part,*
> *Nor his own vision holds what it doth catch….*
> —William Shakespeare, Sonnet 113

If you don't already understand OOP, you won't learn it here. And if you don't already understand OOP *in Ruby,* you probably won't learn it here, either. If you're rusty on those concepts, you can scan Chapter 1, "Ruby in Review," where we cover it rapidly (or you can look at another book).
On the other hand, much of this material is tutorial oriented and fairly elementary. So it will be of some value to the beginner and perhaps less value to the intermediate Ruby programmer. We maintain that a book is a random-access storage device so that you can easily skip the parts that don't interest you.

Using Multiple Constructors

There is no real constructor in Ruby as there is in C++ or Java. The concept is certainly there because objects have to be instantiated and initialized; but the behavior is somewhat different.
In Ruby, a class has a class method `new`, which is used to instantiate new objects. The `new` method calls the user-defined special method `initialize`, which then initializes the attributes of the object appropriately, and `new` returns a reference to the new object.
But what if we want to have multiple constructors for an object? How should we handle that? Nothing prevents us from creating additional class methods that return new objects. Listing 5.1 shows a contrived example in which a rectangle can have two side lengths and three color values. We create additional class methods that assume certain defaults for some of the parameters. (For example, a square is a rectangle with all sides the same length.)

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 5.1. Multiple Constructors

```
class ColoredRectangle

  def initialize(r, g, b, s1, s2)
    @r, @g, @b, @s1, @s2 = r, g, b, s1, s2
  end

  def ColoredRectangle.whiteRect(s1, s2)
    new(0xff, 0xff, 0xff, s1, s2)
  end

   def ColoredRectangle.grayRect(s1, s2)
     new(0x88, 0x88, 0x88, s1, s2)
   end

   def ColoredRectangle.coloredSquare(r, g, b, s)
     new(r, g, b, s, s)
   end
   def ColoredRectangle.redSquare(s)
     new(0xff, 0, 0, s, s)
   end

   def inspect
     "#@r #@g #@b #@s1 #@s2"
   end
 end


 a = ColoredRectangle.new(0x88, 0xaa, 0xff, 20, 30)
 b = ColoredRectangle.whiteRect(15,25)
 c = ColoredRectangle.redSquare(40)
```

So we can define any number of methods we want that create objects according to various specifications. Whether the term *constructor* is appropriate here is a question that we will leave to the language lawyers.

Creating Instance Attributes

An instance attribute in Ruby is always prefixed by an `@` sign. It is like an ordinary variable in that it springs into existence when it is first assigned.

In OO languages, we frequently create methods that access attributes to avoid issues of data hiding. We want to have control over how the internals of an object are accessed from the outside. Typically we use setter and getter methods for this purpose (although in Ruby we don't typically use these terms). These are simply methods used to assign (set) a value or retrieve (get) a value, respectively.

Of course, it is possible to create these functions by hand, as shown here.

```
class Person

  def name
    @name
  end

  def name=(x)
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
    @name = x
  end
  def age
    @age
  end

  # ...

end
```

However, Ruby gives us a shorthand for creating these methods. The `attr` method takes a symbol as a parameter and creates the associated attribute. It also creates a getter of the same name. If the optional second parameter is `true`, it will create a setter as well.

```
class Person
  attr :name, true  # Create @name, name, name=
  attr :age         # Create @age, age
end
```

The related methods `attr_reader`, `attr_writer`, and `attr_accessor` take any number of symbols as parameters. The first will only create `read` methods (to get the value of an attribute); the second will create only `write` methods (to set values); and the third will create both. For example,

```
class SomeClass
  attr_reader :a1, :a2  # Creates @a1, a1, @a2, a2
  attr_writer :b1, :b2  # Creates @b1, a1=, @b2, b2=
  attr_reader :c1, :c2  # Creates @c1, c1, c1=, @c2, c2, c2=
  # ...
end
```

Recall that an assignment to a writer of this form can only be done with a receiver. So within a method, the receiver `self` must be used.

More Elaborate Constructors

As objects grow more complex, they accumulate more attributes that must be initialized when an object is created. The corresponding constructor can be long and cumbersome, forcing us to count parameters and wrap the line past the margin.

One good way to deal with this complexity is to pass in a *block* to the `initialize` method (see Listing 5.2). We can then evaluate the block in order to initialize the object. The trick is to use `instance_eval` instead of `eval` in order to evaluate the block *in the context of the object* rather than that of the caller.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 5.2. A Fancy Constructor

```
class PersonalComputer

  attr_accessor :manufacturer,
    :model, :processor, :clock,
    :ram, :disk, :monitor,
    :colors, :vres, :hres, :net

  def initialize(&block)
    instance_eval &block;
   end

   # Other methods...

end

desktop = PersonalComputer.new do
   self.manufacturer = "Acme"
   self.model = "THX-1138"
   self.processor = "986"
   self.clock = 2.4         # GHz
   self.ram = 1024          # Mb
   self.disk = 800          # Gb
   self.monitor = 25        # inches
   self.colors = 16777216
   self.vres = 1280
   self.hres = 1600
   self.net = "T3"
end

p desktop
```

Several things should be noted here. First of all, we're using accessors for our attributes so that we can assign values to them in an intuitive way. Second, the reference to `self` is necessary because a setter method always takes an explicit receiver to distinguish the method call from an ordinary assignment to a local variable. Of course, rather than define accessors, we could use setter functions.

Obviously, we could perform any arbitrary logic we want inside the body of this block. For example, we could derive certain fields from others by computation.

Also, what if you didn't really want an object to have accessors for each of the attributes? If you prefer, you can use `undef` (at the bottom of the constructor block) to get rid of any or all of these. At the very least, this could prevent accidental assignment of an attribute from outside the object.

Creating Class-level Attributes and Methods

A method or attribute isn't always associated with a specific instance of a class; it can be associated with the class itself. The typical example of a *class method* is the `new` method; it is always invoked in this way because it is called in order to create a new instance (and thus can't belong to any particular instance).

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

We can define class methods of our own if we want. We have already seen this in "Using Multiple Constructors." But their functionality certainly isn't limited to constructors; they can be used for any general-purpose task that makes sense at the class level.
In this next highly incomplete fragment, we assume that we are creating a class to play sound files. The `play` method can reasonably be implemented as an `instance` method; we can instantiate many objects referring to many different sound files. But the `detectHardware` method has a larger context; depending on our implementation, it might not even make sense to create new objects if this method fails. Its context is that of the whole sound-playing environment rather than any particular sound file.

```
class SoundPlayer

  MAX_SAMPLE = 192

  def SoundPlayer.detectHardware
    # ...
  end

  def play
    # ...
  end

end
```

Let's note that there is another way to declare this class method. The following fragment is essentially the same:

```
class SoundPlayer

  MAX_SAMPLE = 192

  def play
    # ...
  end

end

def SoundPlayer.detectHardware
  # ...
end
```

The only difference relates to constants declared in the class. When the class method is declared *outside* of its class declaration, these constants aren't in scope. For example, `detectHardware` in the first fragment can refer directly to `MAX_SAMPLE` if it needs to; in the second fragment, the notation `SoundPlayer::MAX_SAMPLE` would have to be used instead.
Not surprisingly, there are class variables as well as class methods. These begin with a double `@` sign, and their scope is the class rather than any instance of the class.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

The traditional example of using class variables is counting instances of the class as they are created. But they can actually be used for any purpose in which the information is meaningful in the context of the class rather than the object. For a different example, see Listing 5.3.

Listing 5.3. Class Variables and Methods

```
class Metal

  @@current_temp = 70

  attr_accessor :atomic_number

  def Metal.current_temp=(x)
    @@current_temp = x
  end

   def Metal.current_temp
     @@current_temp
   end

   def liquid?
     @@current_temp >= @melting
   end

   def initialize(atnum, melt)
     @atomic_number = atnum
     @melting = melt
   end

end


aluminum = Metal.new(13, 1236)
copper = Metal.new(29, 1982)
gold = Metal.new(79, 1948)
Metal.current_temp = 1600

puts aluminum.liquid?        # true
puts copper.liquid?          # false
puts gold.liquid?            # false

Metal.current_temp = 2100

puts aluminum.liquid?        # true
puts copper.liquid?          # true
puts gold.liquid?            # true
```

Note here that the class variable is initialized at the class level before it is used in a class method. Note also that we can access a class variable from an instance method, but we can't access an instance variable from a class method. After a moment of thought, this makes sense. But what happens if you try? What if we try to print the attribute `@atomic_number` from within the `Metal.current_temp` method? We find that it seems to exist—it doesn't cause an error—but it has the value `nil`. What is happening here?

The answer is that we're not actually accessing the instance variable of class `Metal` at all. We're accessing an instance variable of class `Class` instead. (Remember that in Ruby, `Class` is a class!)

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Such a beast is called a *class instance* variable. We would love to give you a creative example of how to use one, but we can't think of any use for it offhand. We summarize the situation in Listing 5.4.

Listing 5.4. Class and Instance Data

```
class MyClass

  SOME_CONST = "alpha"       # A class-level constant

  @@var = "beta"             # A class variable
  @var = "gamma"             # A class instance variable

  def initialize
    @var = "delta"           # An instance variable
   end

   def mymethod
     puts SOME_CONST          # (the class constant)
     puts @@var               # (the class variable)
     puts @var                # (the instance variable)
   end

   def MyClass.classmeth1
     puts SOME_CONST          # (the class constant)
     puts @@var               # (the class variable)
     puts @var                # (the class instance variable)
   end

 end

 def MyClass.classmeth2
   puts MyClass::SOME_CONST   # (the class constant)
   puts @@var                 # (the class variable)
   puts @var                  # (the class instance variable)
 end


 myobj = MyClass.new
 MyClass.classmeth1           # alpha, beta, gamma
 MyClass.classmeth2           # alpha, beta, gamma
 myobj.mymethod               # alpha, beta, delta
```

We should mention that a class method can be made private with the method `private_class_method`. This works the same way `private` works at the instance level. For additional information refer to "Automatically Defining Class-level Readers and Writers."

Inheriting from a Superclass

We can inherit from a class by using the < symbol:

```
class Boojum < Snark
  # ...
end
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Given this declaration, we can say that the class `Boojum` is a *subclass* of the class `Snark`, or in the same way, `Snark` is a *superclass* of `Boojum`. As we all know, every `Boojum` is a `Snark`, but not every `Snark` is a `Boojum`.

The purpose of inheritance, of course, is to add or enhance functionality. We are going from the more general to the more specific.

As an aside, many languages such as C++ implement *multiple inheritance*. Ruby (like Java and some others) doesn't allow MI, but the mixin facility can compensate for this; see the section "Working with Modules."

Let's look at a (slightly) more realistic example. Suppose that we have a `Person` class and want to create a `Student` class that derives from it. We'll define `Person` this way:

```
class Person

  attr_accessor :name, :age, :sex

  def initialize(name, age, sex)
    @name, @age, @sex = name, age, sex
  end

  # ...

end
```

And we'll then define `Student` in this way:

```
class Student < Person

  attr_accessor :idnum, :hours

  def initialize(name, age, sex, idnum, hours)
    super(name, age, sex)
    @idnum = idnum
    @hours = hours
  end

  # ...

end


# Create two objects
a = Person.new("Dave Bowman", 37, "m")
b = Student.new("Franklin Poole", 36, "m", "000-13-5031", 24)
```

Now let's look at what we've done here. What is this `super` that we see called from `Student`'s `initialize` method? It is simply *acall to the corresponding method in the parent class*. As such, we give it three parameters, whereas our own `initialize` method takes five.

It's not always necessary to use `super` in such a way, but it is often convenient. After all, the attributes of a class form a subset of the attributes of the parent class; so why not use the parent's constructor to initialize them?

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Concerning what inheritance really means, it definitely represents the "is-a" relationship. A `Student` *is-a* `Person`, just as we expect. We'll make three other observations.

First, every attribute (and method) of the parent is reflected in the child. If `Person` had a `height` attribute, `Student` would inherit it; and if the parent had a method named `say_hello`, the child would inherit that, too.

Second, the child can have additional attributes and methods, as you have already seen. That is why the creation of a subclass is often referred to as *extending* a superclass.

Third, the child can *override* or redefine any of the attributes and methods of its parent. This brings up the question of how a method call is resolved. How do I know whether I'm calling the method of this particular class or its superclass?

The short answer is: You don't know, and you don't care. If we invoke a method on a `Student` object, the method for that class will be called *if it exists.* If it doesn't, the method in the superclass will be called, and so on. We say "and so on" because every class (except `Object`) has a superclass.

What if we specifically want to call a superclass method, but we don't happen to be in the corresponding method? We can always create an alias in the subclass before we do anything with it.

```
class Student

  # Assuming Person has a say_hello method...

  alias :say_hi :say_hello

  def say_hello
    puts "Hi, there."
  end

  def formal_greeting
    # Say hello the way my superclass would.
    say_hi
  end

end
```

There are various subtleties relating to inheritance that we don't discuss here, but this is essentially how it works. Be sure to refer to the next section.

Testing Types or Classes of Objects

Frequently we will want to know: What kind of object is this, or how does it relate to this class? There are many ways of making a determination like this.

First of all, the `class` method (that is to say, the instance method named `class`) will always return the class of an object. A synonym is the `type` method.

```
s = "Hello"
n = 237
sc = s.class    # String
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
st = s.type     # String
nc = n.class    # Fixnum
```

Don't be misled into thinking that the thing returned by `class` or `type` is a string representing the class. It is an actual instance of the class `Class`. Thus if we wanted, we could call a `class` method of the target type as though it were an instance method of `Class` (which it is).

```
s2 = "some string"
var = s2.class             # String
my_str = var.new("Hi...")  # A new string
```

We could compare such a variable with a constant class name to see if they were equal; we could even use a variable as the superclass from which to define a subclass. Confused? Just remember that in Ruby, `Class` is an object and `Object` is a class.
Sometimes we want to compare an object with a class to see whether the object belongs to that class. The method `instance_of?` accomplishes this.

```
puts (5.instance_of? Fixnum)        # true
puts ("XYZZY".instance_of? Fixnum)  # false
puts ("PLUGH".instance_of? String)  # true
```

But what if we want to take inheritance relationships into account? The `kind_of?` method (similar to `instance_of?`) takes this issue into account. A synonym is `is_a?` naturally enough because we are describing the classic "is-a" relationship.

```
n = 9876543210
flag1 = n.instance_of? Bignum     # true
flag2 = n.kind_of? Bignum         # true
flag3 = n.is_a? Bignum            # true
flag3 = n.is_a? Integer           # true
flag4 = n.is_a? Numeric           # true
flag5 = n.is_a? Object            # true
flag6 = n.is_a? String            # false
flag7 = n.is_a? Array             # false
```

Obviously, `kind_of` or `is_a?` is more generalized than the `instance_of?` relationship. For an example from everyday life, every dog is a mammal, but not every mammal is a dog. There is one surprise here for the Ruby neophyte. Any module that is mixed in by a class will maintain the "is-a" relationship with the instances. For example, the `Array` class mixes in `Enumerable`; this means that any array is a kind of enumerable entity.

```
x = [1, 2, 3]
flag8 = x.kind_of? Enumerable     # true
flag9 = x.is_a? Enumerable        # true
```

We can also use the numeric relational operators in a fairly intuitive way to compare one class to another. We say intuitive because the less-than operator is used to denote inheritance from a superclass.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
flag1 = Integer < Numeric          # true
flag2 = Integer < Object           # true
flag3 = Object == Array            # false
flag4 = IO >= File                 # true
flag5 = Float < Integer            # false
```

Every class typically has a *relationship operator*=== defined. The expression `class === instance` will be true if the instance belongs to the class. The relationship operator is also known as the *case equality* operator because it is used implicitly in a `case` statement. This is therefore a way to act on the type or class of an expression.
For additional information see the section "Testing Equality of Objects."
We should also mention the `respond_to?` method. This is used when we don't really care what the class is, but just want to know whether it implements a certain method. This, of course, is a rudimentary kind of type information. (In fact, we might say that this is the most important type information of all.) The method is passed a symbol and an optional flag (indicating whether to include private methods in the search).

```
# Search public methods
if wumpus.respond_to?(:bite)
  puts "It's got teeth!"
else
  puts "Go ahead and taunt it."
end

# Optional second parameter will search
# private methods also.

if woozle.respond_to?(:bite,true)
  puts "Woozles bite!"
else
  puts "Ah, the non-biting woozle."
end
```

Sometimes we want to know what class is the immediate parent of an object or class. The instance method `superclass` of class `Class` can be used for this.

```
array_parent = Array.superclass    # Object
fn_parent = 237.class.superclass   # Integer
obj_parent = Object.superclass     # nil
```

Every class except `Object` will have a superclass.

Testing Equality of Objects

> *All animals are equal, but some are more equal than others.*
> —George Orwell, *Animal Farm*

When you write classes, it's convenient if the semantics for common operations are the same as for Ruby's built-in classes. For example, if your classes implement objects that can be

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

ranked, it makes sense to implement the method <=> and mix in the `Comparable` module. Doing so means that all the normal comparison operators work with objects of your class.
However, the picture is less clear when it comes to dealing with object equality. Ruby objects implement five different methods that test for equality. Your classes might end up implementing some of these, so let's look at each in turn.
The most basic comparison is the `equal?` method (that comes from `Object`), which returns `true` if its receiver and parameter have the same object ID. This is a fundamental part of the semantics of objects, and shouldn't be overridden in your classes.
The most common test for equality uses our old friend ==, which tests the values of its receiver with its argument. This is probably the most intuitive test for equality.
Next on the scale of abstraction is the method `eql?`, which is part of `Object`. (Actually, `eql?` is implemented in the `Kernel` module, which is mixed in to `Object`.) Like ==, `eql?` compares its receiver and its argument, but is slightly stricter. For example, different numeric objects will be coerced into a common type when compared using ==, but numbers of different types will never test equal using `eql?`.

```
flag1 = (1 == 1.0)      # true
flag2 = (1.eql?(1.0))   # false
```

The `eql?` method exists for one reason: It is used to compare the values of hash keys. If you want to override Ruby's default behavior when using your objects as hash keys, you'll need to override the methods `eql?` and `hash` for those objects.
Two more equality tests are implemented by every object. The === method is used to compare the target in a case statement against each of the selectors, using *selector*===*target*. Although apparently complex, this rule allows Ruby case statements to be very intuitive in practice. For example, you can switch based on the class of an object:

```
case an_object
  when String
    puts "It's a String."
  when Number
    puts "It's a Number."
  else
    puts "It's something else entirely."
end
```

This works because class `Module` implements === to test whether its parameter is an instance of its receiver or the receiver's parents. So, if `an_object` is the string `"cat"`, the expression `String === an_object` would be `true`, and the first clause in the case statement would fire.
Finally, Ruby implements the match operator =~. Conventionally, this is used by strings and regular expressions to implement pattern matching. However, if you find a use for it in some unrelated classes, you're free to overload it.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

The equality tests == and =~ also have negated forms, != and !~, respectively. These are implemented internally by reversing the sense of the non-negated form. This means that if you implement (say) the method ==, you also get the method != for free.

Controlling Access to Methods

In Ruby, an object is pretty much defined by the interface it provides: the methods it makes available to others. However, when writing a class, you often need to write other, helper methods, used within your class but dangerous if available externally. That is where the `private` method of class `Module` comes in handy.

You can use `private` in two different ways. If you call `private` with no parameters in the body of a class or method definition, subsequent methods will be made private to that class or module. Alternatively, you can pass a list of method names (as symbols) to `private`, and these named methods will be made private. Listing 5.5 shows both forms.

Listing 5.5. Private Methods

```
class Bank
  def openSafe
    # ...
  end
  def closeSafe
    # ...
  end

  private :openSafe, :closeSafe
  def makeWithdrawl(amount)
    if accessAllowed
      openSafe
      getCash(amount)
      closeSafe
    end
  end

  # make the rest private
  private

  def getCash
    # ...
  end
  def accessAllowed
    # ...
  end
end
```

Because the `attr` family of statements effectively just defines methods, attributes are affected by the access control statements such as `private`.

The implementation of `private` might seem strange, but is actually quite clever. Private methods cannot be called with an explicit receiver: They are always called with an implicit receiver of `self`. This means that you can never invoke a `private` method in another object: There is no way to specify that other object as the receiver of the method call. It also means

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

that `private` methods are available to subclasses of the class that defines them, but again only in the same object.

The`protected` access modifier is less restrictive. Protected methods can only be accessed by instances of the defining class and its subclasses. You can specify a receiver with protected methods, so you can invoke those in different objects (as long as they are objects of the same class as the sender). A common use for protected methods is defining accessors to allow two objects of the same type to cooperate with each other. In the following example, objects of class `Person` can be compared based on the person's age, but that age isn't accessible outside the `Person` class.

```
class Person

  def initialize(name, age)
    @name, @age = name, age
  end
  def <=>(other)
    age <=> other.age
  end

  attr_reader :name, :age

  protected   :age

end

p1 = Person.new("fred", 31)
p2 = Person.new("agnes", 43)
compare = (p1 <=> p2)         # -1
x = p1.age                    # Error!
```

To complete the picture, the access modifier `public` is used to make methods public. This shouldn't be a surprise.

As a final twist, normal methods defined outside a class or module definition (that is, the methods defined at the top level) are made `private` by default. Because they are defined in class `Object`, they are globally available, but they cannot be called with a receiver.

Copying an Object

The Ruby built-in methods `Object#clone` and `#dup` produce copies of their receiver. They differ in the amount of context they copy. The `dup` method copies just the object's content, whereas `clone` also preserves things such as singleton classes associated with the object.

```
s1 = "cat"

def s1.upcase
  "CaT"
end

s1_dup   = s1.dup
s1_clone = s1.clone
s1                    #=> "cat"
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
s1_dup.upcase          #=> "CAT"  (singleton method not copied)
s1_clone.upcase        #=> "CaT"  (uses singleton method)
```

Both `dup` and `clone` are *shallow copies:* They copy the immediate contents of their receiver only. If the receiver contains references to other objects, those objects aren't in turn copied; the duplicate simply holds references to them. The following example illustrates this. The object `arr2` is a copy of `arr1`, so changing entire elements, such as `arr2[2]` has no effect on `arr1`. However, both the original array and the duplicate contain a reference to the same `String` object, so changing its contents via `arr2` also affects the value referenced by `arr1`.

```
arr1 = [ 1, "flipper", 3 ]
arr2 = arr1.dup

arr2[2] = 99
arr2[1][2] = 'a'

arr1              # [1, "flapper", 3]
arr2              # [1, "flapper", 99]
```

Sometimes, you want a *deep copy,* where the entire object tree rooted in one object is copied to create the second object. This way, there is guaranteed to be no interaction between the two. Ruby provides no built-in method to perform a deep copy, but there are a couple of techniques you can use to implement one.

The pure way to do it is to have your classes implement a `deep_copy` method. As part of its processing, this method calls `deep_copy` recursively on all the objects referenced by the receiver. You then add a `deep_copy` method to all the Ruby built-in classes that you use. Fortunately, there's a quicker hack using the `Marshal` module. If you use marshaling to dump an object into a string and then load it back into a new object, that new object will be a deep copy of the original.

```
arr1 = [ 1, "flipper", 3 ]
arr2 = Marshal.load(Marshal.dump(arr1))

arr2[2] = 99
arr2[1][2] = 'a'

arr1              # [1, "flipper", 3]
arr2              # [1, "flapper", 99]
```

In this case, notice how changing the string via `arr2` doesn't affect the string referenced by `arr1`.

Working with Modules

There are two basic reasons to use modules in Ruby. The first is simply namespace management; we'll have fewer name collisions if we store constants and methods in modules. A method stored in this way (a module method) is called with the module name; that is,

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

without a real receiver. This is analogous to the way a class method is called. If we see calls such as `File.ctime` and `FileTest.exist?`, we can't tell just from context that `File` is a class and `FileTest` is a module.

The second reason is more interesting: We can use a module as a *mixin*. A mixin is similar to a specialized implementation of multiple inheritance in which only the interface portion is inherited. We've talked about module methods, but what about instance methods? A module isn't a class, so it can't have instances; and an instance method can't be called without a receiver.

As it turns out, a module *can* have instance methods. These become part of whatever class does the `include` of the module.

```
module MyMod

  def meth1
    puts "This is method 1"
  end

end


class MyClass

  include MyMod

  # ...
end


x = MyClass.new
a.meth1                # This is method 1
```

Here `MyMod` is mixed into `MyClass`, and the instance method `meth1` is inherited. You have also seen an `include` done at the top level; in that case, the module is mixed into `Object` as you might expect.

But what happens to our module methods, if there are any? You might think they would be included as class methods, but for whatever reason, Ruby doesn't behave that way. The module methods aren't mixed in.

But we have a trick we can use if we want that behavior. There is a hook called `append_features` that we can override. It is called with a parameter that is the destination class or module (into which this module is being included). For an example of its use, see Listing 5.6.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 5.6. Including a Module with `append_features`

```
module MyMod

  def MyMod.append_features(someClass)
    def someClass.modmeth
      puts "Module (class) method"
    end
    super   # This call is necessary!
  end

   def meth1
     puts "Method 1"
   end

 end


 class MyClass

   include MyMod

   def MyClass.classmeth
     puts "Class method"
   end

   def meth2
     puts "Method 2"
   end

 end


 x = MyClass.new

                        # Output:
 MyClass.classmeth      #   Class method
 x.meth1                #   Method 1
 MyClass.modmeth        #   Module (class) method
 x.meth2                #   Method 2
```

This example is worth examining in detail. First of all, you should understand that `append_features` isn't *just* a hook that is called when an `include` happens; it actually does the work of the `include` operation. That's why the call to `super` is needed; without it, the rest of the module (in this case, `meth1`) wouldn't be included at all.

Also note that within the `append_features` call, there is a method definition. This looks unusual, but it works because the inner method definition is a singleton method (class-level or module-level). An attempt to define an instance method in the same way would result in a `Nested method error`.

Conceivably a module might want to determine the initiator of a mixin. The `append_features` method can also be used for this because the class is passed in as a parameter.

It is also possible to mix in the instance methods of a module as class methods. An example is shown in Listing 5.7.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 5.7. Module Instance Methods Becoming Class Methods

```
class MyMod

  def meth3
    puts "Module instance method meth3"
    puts "can become a class method."
  end

end


class MyClass

   class << self    # Here, self is MyClass
     include MyMod
   end

end


MyClass.meth3

# Output:
#   Module instance method meth3
#   can become a class method.
```

We've been talking about methods. What about instance variables? Although it is certainly possible for modules to have their own instance data, it usually isn't done. However, if you find a need for this capability, nothing is stopping you from using it.

It is possible to mix a module into an object rather than a class (for example, with the `extend` method). See the section "Specializing an Individual Object."

It's important to understand one more fact about modules. It is possible to define methods in your class that will be called by the mixin. This is a very powerful technique that will seem familiar to those who have used Java interfaces.

The classic example (which we've seen elsewhere) is mixing in the `Comparable` module and defining a <=> method. Because the mixed-in methods can call the comparison method, we now have such operators as <, >, <=, and so on.

Another example is mixing in the `Enumerable` module and defining <=> and an iterator `each`. This will give us numerous useful methods such as `collect`, `sort`, `min`, `max`, and `select`.

You can also define modules of your own to be used in the same way. The principal limitation is the programmer's imagination.

Transforming or Converting Objects

Sometimes an object comes in exactly the right form at the right time, but sometimes we need to convert it to something else or pretend it's something it isn't. A good example is the well-known `to_s` method.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Every object can be converted to a string representation in some fashion. But not every object can successfully masquerade as a string. That in essence is the difference between the `to_s` and `to_str` methods. Let's elaborate on that.

Methods such as `puts` and contexts such as `#{...}` interpolation in strings expect to receive a `String` as a parameter. If they don't, they ask the object they *did* receive to convert itself to a `String` by sending it a `to_s` message. This is where you can specify how your object will appear when displayed; simply implement a `to_s` method in your class that returns an appropriate `String`.

```
class Pet

  def initialize(name)
    @name = name
  end

  # ...

  def to_s
    "Pet: #@name"
  end

end
```

Other methods (such as the `String` concatenation operator +) are more picky; they expect you to pass in something that is really pretty close to a `String`. In this case, Matz decided not to have the interpreter call `to_s` to convert nonstring arguments because he felt this would lead to too many errors. Instead, the interpreter invokes a stricter method, `to_str`. Of the built-in classes, only `String` and `Exception` implement `to_str`, and only `String`, `Regexp`, and `Marshal` call it. Typically when you see the runtime error `TypeError: Failed to convert xyz into String`, you know that the interpreter tried to invoke `to_str` and failed.

You can implement `to_str` yourself. For example, you might want to allow numbers to be concatenated to strings:

```
class Numeric

  def to_str
    to_s
  end

end

label = "Number " + 9      # "Number 9"
```

An analogous situation holds for arrays. The method `to_a` is called to convert an object to an array representation, and `to_ary` is called when an array is expected.

An example of when `to_ary` is called is with a multiple assignment. Suppose that we have a statement of this form:

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
a, b, c = x
```

Assuming that `x` were an array of three elements, this would behave in the expected way. But if it isn't an array, the interpreter will try to call `to_ary` to convert it to one. For what it's worth, the method we define can be a singleton (belonging to a specific object). The conversion can be completely arbitrary; here we show an (unrealistic) example in which a string is converted to an array of strings:

```
class String

  def to_ary
    return self.split("")
  end

end

str = "UFO"
a, b, c = str     # ["U", "F", "O"]
```

The `inspect` method implements another convention. Debuggers, utilities such as `irb`, and the debug print method `p` use the `inspect` method to convert an object to a printable representation. If you want classes to reveal internal details when being debugged, you should override `inspect`.

There is another situation in which we'd like to be able to do conversions of this sort under the hood. As a language user, you'd expect to be able to add a `Fixnum` to a `Float`, or divide a `Complex` number by a rational number. However, this is a problem for a language designer. If the `Fixnum` method `+` receives a `Float` as an argument, what can it do? It only knows how to add `Fixnum` values. Ruby implements the `coerce` mechanism to deal with this. When (for example) `+` is passed an argument it doesn't understand, it tries to coerce the receiver and the argument to compatible types and then do the addition based on those types. The pattern for using `coerce` in a class you write is straightforward:

```
class MyNumberSystem

  def +(other)
    if other.kind_of?(MyNumberSystem)
      result = some_calculation_between_self_and_other
      MyNumberSystem.new(result)
    else
      n1, n2 = other.coerce(self)
      n1 + n2
    end
  end

end
```

The value returned by `coerce` is a two-element array containing its argument and its receiver converted to compatible types.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

In this example, we're relying on the type of our argument to perform some kind of coercion for us. If we want to be good citizens, we also need to implement coercion in our class, allowing other types of numbers to work with us. To do this, we need to know the specific types that we can work with directly, and convert ourselves to those types when appropriate. When we can't do that, we fall back on asking our parent.

```
def coerce(other)
  if other.kind_of?(Float)
    return other, self.to_f
  elsif other.kind_of?(Integer)
    return other, self.to_i
  else
    super
  end
end
```

Of course, for this to work, our object must implement `to_i` and `to_f`.
You can use `coerce` as part of the solution for implementing a Perl-like auto-conversion of strings to numbers:

```
class String

  def coerce(n)
    if self['.']
      [n, Float(self)]
    else
      [n, Integer(self)]
    end
  end

end

x = 1 + "23"        # 24
y = 23 * "1.23"     # 29.29
```

We don't necessarily recommend this. But we do recommend that you implement a `coerce` method whenever you are creating some kind of numeric class.

Creating Data-only Classes (`Struct`s)

Sometimes you need to group together a bunch of related data with no other associated processing. You could do this by defining a class:

```
class Address

  attr_accessor :street, :city, :state

  def initialize(street1, city, state)
    @street, @city, @state = street, city, state
  end

end
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
books = Address.new("411 Elm St", "Dallas", "TX")
```

This works, but it's tedious, with a fair amount of repetition. That's why the built-in class `Struct` comes in handy. In the same way that convenience methods such as `attr_accessor` define methods to access attributes, the class `Struct` defines classes that contain just attributes. These classes are *structure templates*.

```
Address = Struct.new("Address", :street, :city, :state)
books = Address.new("411 Elm St", "Dallas", "TX")
```

So, why do we pass the name of the structure to be created in as the first parameter of the constructor, and also assign the result to a constant (`Address`, in this case)?
When we create a new structure template by calling `Struct.new`, a new class is created within class `Struct` itself. This class is given the name passed in as the first parameter, and the attributes given as the rest of the parameters. This means that if we wanted, we could access this newly created class within the namespace of class `Struct`.

```
Struct.new("Address", :street, :city, :state)
books = Struct::Address.new("411 Elm St", "Dallas", "TX")
```

After you've created a structure template, you call its `new` method to create new instances of that particular structure. You don't have to assign values to all the attributes in the constructor: Those that you omit will be initialized to `nil`. Once created, you can access the structure's attributes using normal syntax or by indexing the structure object as if it were a `Hash`. For more information, look up class `Struct` in any reference.
By the way, we advise against the creation of a `Struct` named `Tms` because there is already a predefined `Struct::Tms` class.

Freezing Objects

Sometimes we want to prevent an object from being changed. The `freeze` method (in `Object`) will allow us to do this, effectively turning an object into a constant.
After we freeze an object, an attempt to modify it results in a `TypeError`. Listing 5.8 shows a pair of examples.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 5.8. Freezing an Object

```
str = "This is a test. "
str.freeze

begin
  str << " Don't be alarmed."   # Attempting to modify
rescue => err
  puts "#{ err.class}  #{ err} "
end

 arr = [1, 2, 3]
 arr.freeze

 begin
   arr << 4                     # Attempting to modify
 rescue => err
   puts "#{ err.class}  #{ err} "
 end
# Output:
#   TypeError: can't modify frozen string
#   TypeError: can't modify frozen array
```

However, bear in mind that `freeze` operates on an object reference, not on a variable! This means that any operation resulting in a new object will work. Sometimes this isn't intuitive.

```
str = "counter-"
str.freeze
str += "intuitive"       # "counter-intuitive"

arr = [8, 6, 7]
arr.freeze
arr += [5, 3, 0, 9]      # [8, 6, 7, 5, 3, 0, 9]
```

Why does this happen? A statement `a += x` is semantically equivalent to `a = a + x`. The expression `a + x` is evaluated to a new object, which is then assigned to `a`. The object isn't changed, but the variable now refers to a new object. All the reflexive assignment operators will exhibit this behavior, as will some other methods. Always ask yourself whether you are creating a new object or modifying an existing one; then `freeze` won't surprise you. There is a method `frozen?`, which will tell you whether an object is frozen.

```
hash = {  1 => 1, 2 => 4, 3 => 9 }
hash.freeze
arr = hash.to_a
puts hash.frozen?                  # true
puts arr.frozen?                   # false
hash2 = hash
puts hash2.frozen?                 # true
```

As we see here (with `hash2`), it is the object, not the variable, that is frozen.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

More Advanced Techniques

Not everything in Ruby OOP is straightforward. Some techniques are more complex than others, and some are rarely used. The dividing line will be different for each programmer. We've tried to put items in this part of the chapter that were slightly more involved or slightly more rare in terms of usage.

From time to time, you might ask yourself whether it's possible to do some task or other in Ruby. The short answer is that Ruby is a rich dynamic OOP language with a good set of reasonably orthogonal features; and if you want to do something that you're used to in another language, you can *probably* do it in Ruby.

As a matter of fact, all Turing-complete languages are pretty much the same from a theoretical standpoint. The whole field of language design is the search for a meaningful, convenient notation. Those of you who doubt the importance of a convenient notation should try writing a LISP interpreter in COBOL or doing long division with Roman numerals. Of course, we won't say that *every* language task is elegant or natural in Ruby. Someone would quickly prove us wrong if we made that assertion.

This section also touches on the use of Ruby in various advanced programming styles such as functional programming and aspect-oriented programming. We don't claim expertise in these areas; we are only reporting what other people are saying. Take it all with a grain of salt.

Sending an Explicit Message to an Object

In a static language, you take it for granted that when you call a function, that function name is hard-coded into the program; it is part of the program source. In a dynamic language, we have more flexibility than that.

Every time you invoke a method, you're sending a message to an object. Most of the time, these messages are hard-coded as in a static language, but they need not always be. We can write code that determines at runtime which method to call. The `send` method will allow us to use a `Symbol` to represent a method name.

For an example, suppose that we had an array of objects we wanted to sort, and we wanted to be able to use different fields as sort keys. That's not a problem; we can easily write customized sort blocks. But suppose that we wanted to be a little more elegant and write only a single routine that could sort based on whatever key we specified. Listing 5.9 shows an example.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 5.9. Sorting by Any Key

```
class Person

  attr_reader :name,
    :age,
    :height

  def initialize(name, age, height)
    @name, @age, @height = name, age, height
  end

   def inspect
     "#@name #@age #@height"
   end
end


class Array

  def sort_by(sym)
    self.sort { |x,y| x.send(sym) <=> y.send(sym) }
  end

end


people = []
people << Person.new("Hansel", 35, 69)
people << Person.new("Gretel", 32, 64)
people << Person.new("Ted", 36, 68)
people << Person.new("Alice", 33, 63)

p1 = people.sort_by(:name)
p2 = people.sort_by(:age)
p3 = people.sort_by(:height)

p p1   # [Alice 33 63, Gretel 32 64, Hansel 35 69, Ted 36 68]
p p2   # [Gretel 32 64, Alice 33 63, Hansel 35 69, Ted 36 68]
p p3   # [Alice 33 63, Gretel 32 64, Ted 36 68, Hansel 35 69]
```

Of course, if you really want efficient sorting, this example is incomplete. But it illustrates the example of calling a method whose identity isn't known before runtime.
We'll also mention the alias `__send__`, which does exactly the same thing. It is given this peculiar name, of course, because `send` is a name that might be used (purposely or accidentally) as a user-defined method name.

Specializing an Individual Object

> *I'm a Solipsist, and I must say I'm surprised there aren't more of us.*
> —Letter received by Bertrand Russell

In most object-oriented languages, all objects of a particular class share the same behavior. The class acts as a template, producing an object with the same interface each time the constructor is called.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Although Ruby acts the same way, that isn't the end of the story. When you have a Ruby object, you can change its behavior on-the-fly. Effectively, you're giving that object a private, anonymous subclass: All the methods of the original class are available, but you've added additional behavior for just that object. Because this behavior is private to the associated object, it can only occur once. A thing occurring only once is called a *singleton,* as in *singleton methods* and *singleton classes.*

The word singleton can be confusing because it is also used in a different sense as the name of a well-known design pattern for a class that can only be instantiated once. For this usage, refer to the `singleton.rb` library.

Here we see a pair of objects, both of which are strings. For the second one, we will add a method `upcase` that will override the existing method of that name.

```
a = "hello"
b = "goodbye"

def b.upcase      # create single method
  gsub(/(.)(.)/) {  $1.upcase + $2 }
end

puts a.upcase   # HELLO
puts b.upcase   # GoOdBye
```

Adding a singleton method to an object creates a singleton class for that object if one doesn't already exist. This singleton class's parent will be the object's original class. (This could be considered an anonymous subclass of the original class.) If you want to add multiple methods to an object, you can create the singleton class directly.

```
b = "goodbye"

class << b

  def upcase      # create single method
    gsub(/(.)(.)/) {  $1.upcase + $2 }
  end

  def upcase!
    gsub!(/(.)(.)/) {  $1.upcase + $2 }
  end

end

puts b.upcase  # GoOdBye
puts b         # goodbye
b.upcase!
puts b         # GoOdBye
```

As an aside, we'll note that the more primitive objects (such as a `Fixnum`) cannot have singleton methods added. This is because an object of this nature is stored as an immediate value rather than as an object reference. However, we expect this functionality to be added in a future revision of Ruby.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

If you read some of the library code, you're bound to come across an idiomatic use of singleton classes. Within class definitions, you might see something like this:

```
class SomeClass

  # Stuff...

  class << self
    # more stuff...
  end

  # ... and so on.

end
```

Within the body of a class definition, `self` is the class you're defining, so creating a singleton based on it modifies the class's class. At the simplest level, this means that instance methods in the singleton class are class methods externally.

```
class TheClass
  class << self
    def hello
      puts "hi"
    end
  end
end

# invoke a class method
TheClass.hello            # hi
```

Another common use of this technique is to define class-level helper functions, which we can then access in the rest of the class definition. As an example, we want to define several accessor functions that always convert their results to a string. We could do this by coding each individually. A neater way might be to define a class-level function `accessor_string` that generates these functions for us (as shown in Listing 5.10).

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 5.10. A Class-level Method `accessor_string`

```
class MyClass

  class << self
    def accessor_string(*names)
      names.each do |name|
        class_eval <<-EOD
          def #{ name}
            @#{ name} .to_s
          end
        EOD
      end
    end

  end

  def initialize
    @a = [ 1, 2, 3 ]
    @b = Time.now
  end

  accessor_string :a, :b

end


o = MyClass.new
puts o.a           # 123
puts o.b           # Mon Apr 30 23:12:15 CDT 2001
```

More imaginative examples are left up to you.

The `extend` method will mix a module into an object. The instance methods from the module become instance methods for the object. Let's look at Listing 5.11.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 5.11. Using `extend`

```
module Quantifier

def any?
  self.each {  |x| return true if yield x }
  false
end

def all?
  self.each {  |x| return false if not yield x }
   true
end
end


list = [1, 2, 3, 4, 5]


list.extend(Quantifier)

flag1 = list.any? { |x| x > 5 }          # false
flag2 = list.any? { |x| x >= 5 }         # true
flag3 = list.all? { |x| x <= 10 }        # true
flag4 = list.all? { |x| x % 2 == 0 }     # false
```

In this example, the `any?` and `all?` methods are mixed into the `list` array.

Nesting Classes and Modules

We'll point out that it's possible to nest classes and modules arbitrarily. The programmer new to Ruby might not know this.

Mostly this is for namespace management. Note that the `File` class has a `Stat` class embedded inside it. This helps to encapsulate the `Stat` class inside a class of related functionality, and also allows for a future class named `Stat`, which won't conflict with that one (perhaps a statistics class, for instance).

The `Struct::Tms` class is a similar example. Any new `Struct` is placed in this namespace so as not to pollute the one above it, and `Tms` is really just another `Struct`.

It's also conceivable that you might want to create a nested class simply because the outside world doesn't need that class or shouldn't access it. In other words, you can create classes that are subject to the principle of data hiding just as the instance variables and methods are subject to the same principle at a lower level.

```
class BugTrackingSystem

  class Bug
    #...
  end

  #...

end
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
# Nothing out here knows about Bug.
```

You can nest a class within a module, a module within a class, and so on. If you find interesting and creative uses for this technique, let us all know about it.

Creating Parametric Classes

> *Learn the rules; then break them.*
> —Basho

Suppose that we wanted to create multiple classes that differed only in the initial values of the class-level variables. Recall that a class variable is typically initialized as a part of the class definition.

```
class Terran

  @@home_planet = "Earth"

  def Terran.home_planet
    @@home_planet
  end

  def Terran.home_planet=(x)
    @@home_planet = x
  end

  #...

end
```

That is all fine, but suppose that we had a number of similar classes to define? The novice will think, "Ah, I'll just define a superclass." (See Listing 5.12.)

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 5.12. Parametric Classes #1

```
class IntelligentLife   # Wrong way to do this!

  @@home_planet = nil

  def IntelligentLife.home_planet
    @@home_planet
  end

  def IntelligentLife.home_planet=(x)
     @@home_planet = x
   end
#...

 end

 class Terran < IntelligentLife
   @@home_planet = "Earth"
   #...
 end

 class Martian < IntelligentLife
   @@home_planet = "Mars"
   #...
 end
```

But this won't work. If we call `Terran.home_planet`, we expect a result of `"Earth"`—but we get `"Mars"`!

Why would this happen? The answer is that class variables aren't *truly* class variables; they belong not to the class, but to the entire inheritance hierarchy. The class variables aren't copied from the parent class, but are shared with the parent (and thus with the sibling classes).

We could eliminate the definition of the class variable in the base class; but then the class methods we define would no longer work!

We *could* fix this by moving these definitions to the child classes, but now we've defeated our whole purpose. We're declaring separate classes without any parameterization.

We'll offer a different solution. We'll defer the evaluation of the class variable until runtime by using the `class_eval` method. Listing 5.13 shows a complete solution.

Listing 5.13. Parametric Classes #2

```
class IntelligentLife

  def IntelligentLife.home_planet
    class_eval("@@home_planet")
  end

  def IntelligentLife.home_planet=(x)
    class_eval("@@home_planet = #{ x} ")
  end

   #...
end

class Terran < IntelligentLife
  @@home_planet = "Earth"
  #...
end

class Martian < IntelligentLife
  @@home_planet = "Mars"
  #...
end


puts Terran.home_planet            # Earth
puts Martian.home_planet           # Mars
```

It goes without saying that inheritance still operates normally here. Any instance methods or instance variables defined within `IntelligentLife` will be inherited by `Terran` and `Martian` just as you would expect.

As a minor variation on this theme, we present a slightly different way of doing the same thing (see Listing 5.14). Here we have overridden the `Class.new` method to create a parametric class for us; we inherit from the specified base class and then do a `class_eval` of the block passed in.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 5.14. Parametric Classes #3

```
class Class

  def initialize(klass, &block)
    block = Proc.new
    class_eval(&block)
  end

end


class IntelligentLife

  def IntelligentLife.home_planet
    class_eval("@@home_planet")
  end

  def IntelligentLife.home_planet=(x)
    class_eval("@@home_planet = #{ x} ")
  end

  #...

end

Terran = Class.new(IntelligentLife) do
   @@home_planet = "Earth"
end

Martian = Class.new(IntelligentLife) do
   @@home_planet = "Mars"
end
```

This technique resembles the fancy constructor of the section "More Elaborate Constructors." The principle is the same, but we are working at the class level rather than the instance level. (Of course, `Class` is an object, so we can still regard it as the instance level if we want.)

We should mention that there are other ways of doing this. Use your creativity.

Using Continuations to Implement a Generator

One of the more abstruse features of Ruby is the *continuation*. This is a structured way of handling a nonlocal jump and return; a continuation object stores a return address and an execution context. It is somewhat analogous to the `setjmp/longjmp` feature in C, but it stores more context.

The `Kernel` method `callcc` takes a block and returns an object of the `Continuation` class. The object returned is also passed into the block as a parameter, just to keep things confusing.

The only method of `Continuation` is `call`, which causes a nonlocal return to the end of the `callcc` block. The `callcc` can be terminated either by falling through the block or by calling the `call` method.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

There is a known shortage of examples of how to use continuations. The best one we have seen comes from Jim Weirich, who implemented a generator as a result of his discussion with another Ruby programmer, Hugh Sasse.

A generator is made possible by `suspend` in Icon (also found in Prolog), which allows a function to resume execution just after the last place it returned a value. Hugh describes it as similar to an inside-out yield.

Listing 5.15, then, is Jim's implementation of a generator that generates Fibonacci numbers one after another. Continuations are used to preserve the call state from one invocation to the next.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 5.15. Fibonacci Generator

```
class Generator

  def initialize
    do_generation
  end

  def next
    callcc {  |here|
      @main_context = here;
       @generator_context.call
    }
   end

   private

   def do_generation
     callcc {  |context|
       @generator_context = context;
       return
     }
     generating_loop
   end

   def generate(value)
     callcc {  |context|
       @generator_context = context;
       @main_context.call(value)
     }
   end
end

# Subclass this and define a generating_loop

class FibGenerator < Generator
  def generating_loop
    generate(1)
    a, b = 1, 1
    loop do
      generate(b)
      a, b = b, a+b
    end
  end
end
 # Now instantiate the class...

 fib = FibGenerator.new

puts fib.next            # 1
puts fib.next            # 1
puts fib.next            # 2
puts fib.next            # 3
puts fib.next            # 5
puts fib.next            # 8
puts fib.next            # 13

# And so on...
```

We can't help but feel that there are practical applications for this idea. If you think of some, share them with us all.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Storing Code as Objects

Not surprisingly, Ruby gives you several alternatives when it comes to storing a chunk of code in the form of an object. In this section, we'll take a look at `Proc` objects, `Method` objects, and `UnboundMethod` objects.

The built-in class `Proc` is used to wrap Ruby blocks in an object. `Proc` objects, like blocks, are closures, and therefore carry around the context in which they were defined.

```
p = Proc.new {  |a| puts "Param is #{ a} " }

p.call(99)          # Param is 99
```

`Proc` objects are also created automatically by Ruby when a method defined with a trailing `&` parameter is called with a block.

```
def take_block(a, &block)
  puts block.type
  a.times { |i| block[i, i*i] }
end

take_block(3) {  |n,s| puts "#{ n}  squared is #{ s} " }
```

This example also shows the use of braces (`{ }`) as an alias for the `call` method. The output is shown here:

```
Proc
0 squared is 0
1 squared is 1
2 squared is 4
```

If you have a `Proc` object, you can pass it to a method that's expecting a block, preceding its name with an `&`, as shown here:

```
p = proc {  |n| print n, "... " }
(1..3).each(&p)                     # 1... 2... 3...
```

Ruby also lets you turn a method into an object. Historically, this is done using `Object#method`, which creates a `Method` object as a closure in a particular object.

```
str = "cat"
meth = str.method(:length)

a = meth.call               #  3  (length of "cat")

str << "erpillar"

b = meth.call               # 11  (length of "caterpillar")

str = "dog"

# Note the next call! The variable str refers to a new object
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
# ("dog") now, but meth is still bound to the old object.

c = meth.call               # 11  (length of "caterpillar")
```

As of Ruby 1.6.2, you can also use `Module#instance_method` to create `UnboundMethod` objects. These represent a method that is associated with a class, rather than one particular object. Before calling an `UnboundMethod` object, you must first bind it to a particular object. This act of binding produces a `Method` object, which you call normally.

```
umeth = String.instance_method(:length)

m1 = umeth.bind("cat")
m1.call                         # 3

m2 = umeth.bind("caterpillar")
m2.call                         # 11
```

This explicit binding makes the `UnboundMethod` object a little more intuitive than `Method`.

Automatically Defining Class-level Readers and Writers

You have seen the methods `attr_reader`, `attr_writer`, and `attr_accessor`, which make it a little easier to define readers and writers (getters and setters) for instance attributes. But what about class-level attributes?

Ruby has no similar facility for creating these automatically. But we can make our own facility by adding to the class `Class`. In Listing 5.16 we name them similarly, only prefixing the names with a *c* for *class*.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 5.16. A Shorthand for Creating Class Attributes

```
class Class

  def cattr_reader(*syms)
    syms.each do |sym|
      class_eval <<-EOS
        if ! defined? @@#{ sym.id2name}
          @@#{ sym.id2name}  = nil
        end
        def self.#{ sym.id2name}
           @@#{ sym}
         end
       EOS
     end
   end

   def cattr_writer(*syms)
     syms.each do |sym|
       class_eval <<-EOS
         if ! defined? @@#{ sym.id2name}
           @@#{ sym.id2name}  = nil
         end
         def self.#{ sym.id2name} =(obj)
           @@#{ sym.id2name}  = obj
         end
       EOS
     end
   end

   def cattr_accessor(*syms)
     cattr_reader(*syms)
     cattr_writer(*syms)
   end

end
class MyClass

   @@alpha = 123                # Initialize @@alpha

   cattr_reader :alpha          # MyClass.alpha()
   cattr_writer :beta           # MyClass.beta=()
   cattr_accessor :gamma        # MyClass.gamma() and
                                #   MyClass.gamma=()

   def MyClass.look
     puts "#@@alpha, #@@beta, #@@gamma"
   end

   #...

end


puts MyClass.alpha             # 123
MyClass.beta = 456
MyClass.gamma = 789
puts MyClass.gamma             # 789

MyClass.look                   # 123, 456, 789
```

Most classes are no good without instance level data. We've only omitted it from Listing 5.16 for clarity.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Working in Advanced Programming Disciplines

Brother, can you paradigm?
—Graffiti seen at IBM Austin, 1989

Many philosophies of programming are popular in various circles. These are often difficult to characterize in relation to object-oriented or dynamic techniques; and some of these styles can be actually independent of whether a language is dynamic or object-oriented.
Because we are far from experts in these matters, we are relying mostly on hearsay. So take these next paragraphs with a grain of sodium chloride.
Some programmers prefer a flavor of OOP known as *prototype-based* OOP (or *classless* OOP). In this world, an object isn't described as a member of a class. It is built from the ground up, and other objects are created based on the prototype. Ruby has at least rudimentary support for this programming style because it allows singleton methods for individual objects, and the `clone` method will clone these singletons. Interested readers should also look at the simple `Ostruct` class for building Python-like objects; and you should also be aware of how `method_missing` works.
One or two limitations in Ruby hamper classless OOP. Certain objects such as `Fixnum`s are stored not as references, but as immediate values so that they can't have singleton methods. This is supposed to change in the future; but at the time of this writing, it's impossible to project when it will happen.
In *functional programming* (*FP*), emphasis is placed on the evaluation of expressions rather than the execution of commands. An FP language is one that encourages and supports functional programming, and as such, there is a natural gray area. Nearly all would agree that Haskell is a *pure* functional language, whereas Ruby certainly is not.
But Ruby has at least some minimal support for FP; it has a fairly rich set of methods for operating on arrays (lists), and it has `Proc` objects so that code can be encapsulated and called over and over. Ruby allows the method chaining that is so common in FP; although it is easy to be bitten by the phenomenon of a bang method (such as `sort!` or `gsub!`) that returns `nil` when the receiver doesn't actually change.
There have been some initial efforts at a library that would serve as a kind of FP compatibility layer, borrowing certain ideas from Haskell. At the time of this writing, these efforts aren't complete.
The concept of *aspect-oriented programming(AOP)* is an interesting one. In AOP, we try to deal with programming issues that crosscut the modular structure of the program. In other words, some activities or features of a system will be scattered across the system in code fragments here and there, rather than being gathered into one tidy location. We are attempting to modularize things that in traditional OOP or procedural techniques are difficult to modularize. We are working at right angles to our usual way of thinking.
Ruby certainly wasn't created specifically with AOP in mind. But it was designed to be a very flexible and dynamic language, and it is conceivable that these techniques can be facilitated

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

by a library. In fact, there is a library called `AspectR`, which is an early effort at implementing AOP; see the Ruby Application Archive for the most recent version.

The concept of *Design by Contract(DBC)* is well-known to Eiffel devotees, although it is certainly known outside those circles as well. The general idea is that a method or class implements a contract; certain pre-conditions must be true if it is going to do its job, and it guarantees that certain post-conditions are true afterward. The robustness of a system can be greatly enhanced by the ability to specify this contract explicitly and have it automatically checked at runtime. The usefulness of the technique is expanded by the inheritance of contract information as classes are extended.

The Eiffel language has DBC explicitly built in; Ruby doesn't. There are at least two usable implementations of DBC libraries, however, and we recommend that you choose one and learn it.

*Design patterns*have inspired much discussion over the last few years. These, of course, are highly language-independent and can be implemented well in many languages. But again, Ruby's unusual flexibility makes them perhaps more practical than in some other environments. Well-known examples of these are given elsewhere; the *Visitor* pattern is essentially implemented in Ruby's default iterator `each`, and other patterns are part of Ruby's standard distribution (see `delegator.rb` and `singleton.rb`).

The *Extreme Programming(XP)*discipline is gaining devotees daily. This methodology encourages (among other things) early testing and refactoring on-the-fly.

XP isn't language specific, although it might be easier in some languages than others. Certainly we maintain that Ruby makes refactoring easier than many languages would, although that is a highly subjective claim. But the existence of the `RubyUnit` library is what makes for a real blending of Ruby and XP. This library facilitates unit testing; it is powerful, easy to use, and it has proven useful in developing other Ruby software in current use. We highly recommend the XP practice of testing early and often, and we recommend `RubyUnit` for those who want to do this in Ruby.

We should also mention Lapidary, another unit-testing framework created by XP fan Nathaniel Talbott (coming from a Smalltalk perspective). It can be found in the Ruby Application Archive.

By the time you read this, many of the issues we talk about in this section will have been fleshed out more. As always, your two best resources for the latest information are the `comp.lang.ruby` newsgroup and the Ruby Application Archive.

Working with Dynamic Features

> *Skynet became self-aware at 2:14 a.m. EDT August 29, 1997.*
> —Terminator 2: Judgment Day

Many of you will come from the background of a very static language such as C. To those readers, we will address this rhetorical question: Can you imagine writing a C function that will take a string, treat it as a variable name, and return the value of the variable?

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

No? Then can you imagine removing or replacing the definition of a function? Can you imagine trapping calls to nonexistent functions? Or determining the name of the calling function? Or automatically obtaining a list of user-defined program elements (such as a list of all your functions)?
Ruby makes this sort of thing possible. This runtime flexibility, the ability to examine and change program elements at runtime, makes many problems easier. A runtime tracing utility, a debugger, and a profiling utility are all easy to write *for* Ruby and *in* Ruby. The well-known programs `irb` and `xmp` both use dynamic features of Ruby in order to perform their magic. These abilities take getting used to, and they are easy to abuse. But the concepts have been around for many years (they are at least as old as LISP) and are regarded as tried and true in the Scheme and Smalltalk communities as well. Even Java, which owes so much to C and C ++, has some dynamic features; so we expect this way of thinking to increase in popularity as time goes by.

Evaluating Code Dynamically

The global function `eval` compiles and executes a string that contains a fragment of Ruby code. This is a powerful (if slightly dangerous) mechanism because it allows you to build up code to be executed at runtime. For example, the following code reads in lines of the form *name = expression;* it then evaluates each expression and stores the result in a hash indexed by the corresponding variable name.

```
parameters = { }

ARGF.each do |line|
  name, expr = line.split(/\s*=\s*/, 2)
  parameters[name] = eval expr
end
```

Suppose that the input contained these lines.

```
a = 1
b = 2 + 3
c = `date`
```

Then the hash `parameters` would end up with the value `{ "a"=>1, "b"=>5, "c"=>"Mon Apr 30 21:17:47 CDT 2001\n"}`. This example also illustrates the danger of `eval`ing strings when you don't control their contents; a malicious user could put `d=`rm *`` in the input and ruin your day.
Ruby has three other methods that evaluate code on-the-fly: `class_eval`, `module_eval`, and `instance_eval`. The first two are synonyms, and all three do effectively the same thing; they evaluate a string or a block, but while doing so, they change the value of `self` to their own receiver. Perhaps the most common use of `class_eval` allows you to add methods to a class when all you have is a reference to that class. We used this in the `hook_method` code in the `Trace` example in the section "Tracking Changes to

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

a Class or Object Definition." You'll find other examples in the more dynamic library modules, such as `delegate.rb`.

The `eval` method also makes it possible to evaluate local variables in a context outside their scope. We don't advise doing this lightly, but it's nice to have the capability.

Ruby associates local variables with blocks, with high level definition constructs (class, module, and method definitions), and with the top-level of your program (the code outside any definition constructs). Associated with each of these scopes is the binding of variables, along with other housekeeping details. Probably the ultimate user of bindings is `irb`, the interactive Ruby shell, which uses bindings to keep the variables in the program that you type separate from its own.

You can encapsulate the current binding in an object using the method `Kernel#binding`. Having done that, you can pass the binding as the second parameter to `eval`, setting the execution context for the code being evaluated.

```
def aMethod
  a = "local variable"
  return binding
end

the_binding = aMethod
eval "a", the_binding   # "local variable"
```

Interestingly, the presence of a block associated with a method is stored as part of the binding, enabling tricks such as this:

```
def aMethod
  return binding
end

the_binding = aMethod {  puts "hello" }
eval "yield", the_binding                # hello
```

Removing Definitions

The dynamic nature of Ruby means that pretty much anything that can be defined can also be undefined. One conceivable reason to do this is to decouple pieces of code that are in the same scope by getting rid of variables once they have been used; another reason might be to specifically disallow certain dangerous method calls. Whatever your reason for removing a definition, it should naturally be done with caution because it can conceivably lead to debugging problems.

The radical way to undefine something is with the `undef` keyword (not surprisingly, the opposite of `def`). You can `undef` methods, local variables, and constants at the top level. Although a classname is a constant, you *cannot* remove a class definition this way.

```
def asbestos
  puts "Now fireproof"
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
end

tax = 0.08

PI = 3

asbestos
puts "PI=#{ PI} , tax=#{ tax} "

undef asbestos
undef tax
undef PI

# Any reference to the above three
# would now give an error.
```

Within a class definition, a method or constant can be undefined in the same context in which it was defined. You can't `undef` within a method definition or `undef` an instance variable. We also have the `remove_method` and `undef_method` methods available to us (defined in `Module`). The difference is slightly subtle: `remove_method` will remove the current (or nearest) definition of the method, whereas `undef_method` will literally cause the method to be undefined (removing it from superclasses as well). Listing 5.17 is an illustration of this.

Listing 5.17. Removing and Undefining Methods

```
class Parent

  def alpha
    puts "parent alpha"
  end

  def beta
    puts "parent beta"
  end

end

class Child < Parent
   def alpha
     puts "child alpha"
   end

   def beta
     puts "child beta"
   end

   remove_method :alpha   # Remove "this" alpha
   undef_method :beta     # Remove every beta

end


x = Child.new

x.alpha          # parent alpha
x.beta           # Error!
```

The `remove_const` method will remove a constant.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
module Math

  remove_const :PI

end

# No PI anymore!
```

Note that it is possible to remove a class definition in this way (because a class identifier is simply a constant).

```
class BriefCandle
  #...
end

out_out = BriefCandle.new

class Object
  remove_const :BriefCandle
end

# Can't instantiate BriefCandle again!
# (Though out_out still exists...)
```

Note that methods such `remove_const` and `remove_method` are (naturally enough) private methods. That is why we show them being called from inside a class or module definition rather than outside.

Obtaining Lists of Defined Entities

The reflection API of Ruby enables us to examine the classes and objects in our environment at runtime. We'll look at methods defined in `Module`, `Class`, and `Object`.
The `Module` module has a method named `constants` that returns an array of all the constants in the system (including class and module names). The `nesting` method returns an array of all the modules nested at the current location in the code.
The instance method `Module#ancestors` will return an array of all the ancestors of the specified class or module.

```
list = Array.ancestors
# [Array, Enumerable, Object, Kernel]
```

The `constants` method will list all the constants accessible in the specified module. Any ancestor modules are included.

```
list = Math.constants    # ["E", "PI"]
```

The `class_variables` method will return a list of all class variables in the given class and its superclasses. The `included_modules` method will list the modules included in a class.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
class Parent
  @@var1 = nil
end

class Child < Parent
  @@var2 = nil
end

list1 = Parent.class_variables   # ["@@var1"]
list2 = Array.included_modules   # [Enumerable, Kernel]
```

The `Class` methods `instance_methods` and `public_instance_methods` are synonyms; they return a list of the public instance methods for a class. The methods `private_instance_methods` and `protected_instance_methods` behave as expected. Any of these can take a Boolean parameter, which defaults to `false`; if it is set to `true`, superclasses will be searched as well, resulting in a larger list.

```
n1 = Array.instance_methods.size                # 66
n2 = Array.public_instance_methods.size         # 66
n3 = Array.private_instance_methods.size        # 1
n4 = Array.protected_instance_methods.size      # 0
n5 = Array.public_instance_methods(true).size   # 106
```

The `Object` class has a number of similar methods that operate on instances (see Listing 5.18). `methods` and `public_methods` are synonyms that return a list of publicly accessible methods. The methods `private_methods`, `protected_methods`, and `singleton_methods` all behave as expected.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 5.18. Reflection and Instance Variables

```
class SomeClass

  def initialize
    @a = 1
    @b = 2
  end

  def mymeth
    #...
   end

   protected :mymeth

 end


 x = SomeClass.new

 def x.newmeth
   # ...
 end

 iv = x.instance_variables        # ["@b", "@a"]

 meth = x.methods.size            # 37
 pub  = x.public_methods.size     # 37
 pri  = x.private_methods.size    # 66
 pro  = x.protected_methods.size  # 1
 sm   = x.singleton_methods.size  # 1
```

Note that none of the preceding ever takes a parameter.

Examining the Call Stack

> *And you may ask yourself:*
> *Well, how did I get here?*
> —Talking Heads, "Once in a Lifetime"

Sometimes we want to know who our caller was. This could be useful information if, for example, we had a fatal exception. The `caller` method, defined in `Kernel`, makes this possible. It returns an array of strings in which the first element represents the caller, the next element represents the caller's caller, and so on.

```
def func1
  puts caller[0]
end

def func2
  func1
end

func2              # Prints: somefile.rb:6:in `func2'
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

The string is in the form *file;line* or *file:line: in method,* as shown previously.

Monitoring Execution of a Program

A Ruby program can *introspect* or examine its own execution. There are many applications for such an ability; the interested reader can refer to the sources `debug.rb`, `profile.rb`, and `tracer.rb`. It is even conceivable to use this facility in creating a *design-by-contract (DBC)* library—although the most popular one at the time of this writing doesn't use this technique.

The interesting thing is that this trick is implemented purely in Ruby. We use the Ruby method `set_trace_func`, which allows you to invoke a block whenever significant events happen in the execution of a program. A good reference will show the calling sequence for `set_trace_func`, so we'll just show a simple example here.

```
def meth(n)
  sum = 0
  for i in 1..n
    sum += i
  end
  sum
end

set_trace_func proc do |event, file, line,
                        id, binding, klass, *rest|
  printf "%8s %s:%d  %s/%s\n", event, file, line,
                              klass, id
end

meth(2)
```

This produces the output:

```
        line prog.rb:13  false/
        call prog.rb:1  Object/meth
        line prog.rb:2  Object/meth
        line prog.rb:3  Object/meth
      c-call prog.rb:3  Range/each
        line prog.rb:4  Object/meth
      c-call prog.rb:4  Fixnum/+
c-return prog.rb:4  Fixnum/+
    line prog.rb:4  Object/meth
  c-call prog.rb:4  Fixnum/+
c-return prog.rb:4  Fixnum/+
c-return prog.rb:4  Range/each
    line prog.rb:6  Object/meth
  return prog.rb:6  Object/meth
```

Another related method is `Kernel#trace_var`, which invokes a block whenever a global variable is assigned a value.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Suppose that you want to trace the execution of a program from outside, strictly as an aid in debugging. The simplest way to see what a program is doing is to use the `tracer` library that we mentioned previously. Given a program `prog.rb`:

```
def meth(n)
  (1..n).each { |i| puts i}
end

meth(3)
```

You can simply load `tracer` from the command line:

```
% ruby -r tracer prog.rb
#0:prog.rb:1::-:     def meth(n)
#0:prog.rb:1:Module:>:     def meth(n)
#0:prog.rb:1:Module::     def meth(n)
#0:prog.rb:2:Object:-:       sum = 0
#0:prog.rb:3:Object:-:       for i in 1..n
#0:prog.rb:3:Range:>:       for i in 1..n
#0:prog.rb:4:Object:-:         sum += i
#0:prog.rb:4:Fixnum:>:         sum += i
#0:prog.rb:4:Fixnum::         sum += i
#0:prog.rb:4:Fixnum:
```

The lines output by `tracer` show the thread number, the filename and line number, the class being used, the event type, and the line from the source file being executed. The event types include `'-'` when a source line is executed, `'>'` for a call, `'<'` for a return, `'C'` for a class, and `'E'` for an end.

Traversing the Object Space

The Ruby runtime system needs to keep track of all known objects (if for no other reason than to be able to garbage-collect those no longer referenced). This information is made accessible via the `ObjectSpace.each_object` method.

```
ObjectSpace.each_object do |obj|
  printf "%20s: %s\n", obj.class, obj.inspect
end
```

If you specify a class or module as a parameter to `each_object`, only objects of that type will be returned.

The `ObjectSpace` module is also useful in defining object finalizers (see the section "Defining Finalizers for Objects").

Handling Calls to Nonexistent Methods

Sometimes it's useful to be able to write classes that respond to arbitrary method calls. For example, you might want to wrap calls to external programs in a class, providing access to each program as a method call. You can't know ahead of time the names of all these programs,

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

so you can't create the methods as you write the class. Here comes `Object#method_missing` to the rescue. Whenever a Ruby object receives a message for a method that isn't implemented in the receiver, it invokes the `method_missing` method instead. You can use that to catch what would otherwise be an error, treating it as a normal method call. Let's implement the operating system `CommandWrapper` class:

```
class CommandWrapper

  def method_missing(method, *args)
    system(method.to_s, *args)
  end

end


cw = CommandWrapper.new
cw.date                    # Sat Apr 28 22:50:11 CDT 2001
cw.du '-s', '/tmp'         # 166749  /tmp
```

The first parameter to `method_missing` is the name of the method that was called (and that couldn't be found). Whatever was passed in that method call is then given as the remaining parameters.

If your `method_missing` handler decides that it doesn't want to handle a particular call, it should call `super`, rather than raising an exception. That allows `method_missing` handlers in superclasses to have a shot at dealing with the situation. Eventually, the `method_missing` method defined in class `Object` will be invoked, and that ends up raising an exception.

Tracking Changes to a Class or Object Definition

Perhaps we should start this by asking: Who cares? Why are we interested in tracking changes to classes?

One possible reason is that we're trying to keep track of the state of the Ruby program being run. Perhaps we're implementing some kind of GUI-based debugger, and we need to refresh a list of methods if our user adds one on-the-fly.

Another reason might be that we're doing clever things to other classes. For example, say that we wanted to write a module that could be included in any class definition. From then on, any call to a method in that class will be traced. We might use it something like this:

```
class MyClass

  def one
  end

  include Trace

  def two(x, y)
  end
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
end


m = MyClass.new
m.one                 # one called. Params =
m.two(1, 'cat')       # two called. Params = 1, cat
```

It will also work for any subclasses of the class we're tracing:

```
class Fred < MyClass

  def meth(*a)
  end
end

Fred.new.meth(2,3,4,5)    # meth called. Params = 2, 3, 4, 5
```

We could implement this module as shown in Listing 5.19.

Listing 5.19. Trace Module

```
module Trace

  def Trace.append_features(into)
    into.instance_methods.each {  |m| Trace.hook_method(into, m) }
    def into.method_added(method)
      unless @adding
        @adding = true
        Trace.hook_method(self, method)
        @adding = false
       end
     end
     super
   end

   def Trace.hook_method(klass, method)
     klass.class_eval <<-EOD
     alias :old_#{ method}   :#{ method}
     def #{ method} (*args)
       puts "#{ method}  called. Params = #{ args.join(', ')} "
       old_#{ method} (*args)
     end
     EOD
   end

end
```

This code has two main methods. The first, `append_features`, is a callback invoked whenever a module is inserted into a class. Our version does two things. It calls `hook_method` for every method that's already been defined in the target class, and it inserts a definition for `method_added` into that class. This means that any subsequently added method will also be detected and hooked.

The hook itself is pretty straightforward: When a method is added, it is immediately aliased to the name `old_`*`name`*. The original method is then replaced by out tracing code, which dumps out the method name and parameters before invoking the original method.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

To detect the addition of a new class method to a class or module, we can define a class method `singleton_method_added` within that class. (Recall that a singleton method in this sense is what we usually refer to as a class method because `Class` is an object.) This method comes from `Kernel` and by default does nothing, but we can make it behave as we prefer.

```
class MyClass

  def MyClass.singleton_method_added(sym)
    puts "Added method #{ sym.id2name}  to class MyClass."
  end

  def MyClass.meth1
    puts "I'm meth1."
  end

end

def MyClass.meth2
  puts "And I'm meth2."
end
```

The output we get from this is as follows:

```
Added method singleton_method_added to class MyClass.
Added method meth1 to class MyClass.
Added method meth2 to class MyClass.
```

Note that there are actually three methods added here. Perhaps contrary to expectation, `singleton_method_added` is able to track its own addition to the class.
The `inherited` method (from `Class`) is used in much the same way. It is called whenever a class is subclassed by another.

```
class MyClass

  def MyClass.inherited(subclass)
    puts "#{ subclass}  inherits from MyClass."
  end

  # ...

end


class OtherClass < MyClass

  # ...

end

# Output: OtherClass inherits from MyClass.
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

We can also track the addition of a module's instance methods to an object (done via the `extend` method). The method `extend_object` is called whenever an `extend` is done.

```
module MyMod

  def MyMod.extend_object(obj)
    puts "Extending object id #{ obj.id} , type #{ obj.type} "
    super
  end

  # ...

end


x = [1, 2, 3]
x.extend(MyMod)

# Output:
# Extending object id 36491192, type Array
```

Note that the call to `super` is needed in order for the real `extend_object` method to do its work. This is analogous to the behavior of `append_features` (see the section "Working with Modules"). This method can also be used to track the usage of modules.

Defining Finalizers for Objects

Ruby classes have constructors (the methods `new` and `initialize`) but don't have destructors (methods that delete objects). That's because Ruby uses mark-and-sweep garbage collection to remove unreferenced objects; a destructor would make no sense. However, people coming to Ruby from languages such as C++ seem to miss the facility, and often ask how they can write code to handle the finalization of objects. The simple answer is that there is no real way to do it reliably. But you *can* arrange to have code called when an object is garbage-collected.

```
a = "hello"
puts "The string 'hello' has an object id #{ a.id} "
ObjectSpace.define_finalizer(a) {  |id| puts "Destroying #{ id} " }
puts "Nothing to tidy"
GC.start
a = nil
puts "The original string is now a candidate for collection"
GC.start
```

This produces the following output:

```
The string 'hello' has an object id 537684890
Nothing to tidy
The original string is now a candidate for collection
Destroying 537684890
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Note that by the time the finalizer is called, the object has basically been destroyed already. An attempt to convert the ID you receive back into an object reference using `ObjectSpace._id2ref` will raise a `RangeError`, complaining that you are attempting to use a recycled object.
However, all this might be moot. There's a style of programming in Ruby that uses blocks to encapsulate the use of a resource. At the end of the block, the resource is deleted and life carries on merrily, all without the use of finalizers. For example, consider the block form of `File.open`:

```
File.open("myfile.txt") do |aFile|
  l1 = aFile.read
  # ...
end
```

Here the `File` object is passed into the block. When the block exits, the file is closed, all under control of the `open` method. If you wanted to write a subset of `File.open` in Ruby (for efficiency, it's currently written in C as part of the runtime system), it might look something like this:

```
def File.open(name, mode = "r")
  f = os_file_open(name, mode)
  if block_given?
    begin
      yield f
    ensure
      f.close
    end
    return nil
  else
    return f
  end
end
```

The routine tests for the presence of a block. If found, it invokes that block, passing in the open file. It does this in the context of a `begin-end` block, ensuring that it will close the file after the block terminates, even if an exception is thrown.

Dynamically Instantiating a Class by Name

We have seen this question more than once. Given a string containing the name of a class, how can we create an instance of that class?
The answer is annoyingly simple. Use `eval` for the purpose.

```
classname = "Array"

classvar = eval(classname)
x = classvar.new(4, 1)        # [1, 1, 1, 1]
```

As always, make sure that the string you're evaluating is a safe one.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Summary

You've seen here a few of the more esoteric or advanced techniques in OOP, as well as some of the more everyday usages. We've also looked at Ruby's reflection API, some interesting consequences of Ruby's dynamic nature, and various neat tricks that can be done in a dynamic language.

It's time now to rejoin the real world. After all, OOP is not an end in itself, but a means to an end; the end is to write applications that are effective, bug free, and maintainable.

In modern computing, these applications frequently need a graphical interface. In Chapter 6, "Graphical Interfaces for Ruby," we discuss creating graphical interfaces in Ruby.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17