Classes

Chapter 9

So far we've seen several different kinds, or classes, of objects: strings, integers, floats, arrays, and a few special objects (true, false, and nil) which we'll talk about later. In Ruby, these classes are always capitalized: String, Integer, Float, Array... etc. In general, if we want to create a new object of a certain class, we use new:

```
a = Array.new  + [12345]  # Array  addition.
b = String.new + 'hello'  # String addition.
c = Time.new

puts 'a = '+a.to_s
puts 'b = '+b.to_s
puts 'c = '+c.to_s
```

```
a = [12345]
b = hello
c = 2016-11-17 01:26:50 -0600
```

Because we can create arrays and strings using [...] and '...' respectively, we rarely create them using new. (Though it's not really obvious from the above example, String.new creates an empty string, and Array.new creates an empty array.) Also, numbers are special exceptions: you can't create an integer with Integer.new. You just have to write the integer.

The Time Class

So what's the story with this Time class? Time objects represent moments in time. You can add (or subtract) numbers to (or from) times to get new times: adding 1.5 to a time makes a new time one-and-a-half seconds later:

```
time  = Time.new   # The moment I generated this web page.
time2 = time + 60  # One minute later.

puts time
puts time2
```

```
2016-11-17 01:26:50 -0600
2016-11-17 01:27:50 -0600
```

You can also make a time for a specific moment using Time.mktime:

```
puts Time.mktime(2000, 1, 1)          # Y2K.
```

```
puts Time.mktime(1976, 8, 3, 10, 11)  # When I was born.
```

```
2000-01-01 00:00:00 -0600
1976-08-03 10:11:00 -0500
```

Notice: that's when I was born in Pacific Daylight Savings Time (PDT). When Y2K struck, though, it was Pacific Standard Time (PST), at least to us West Coasters. The parentheses are to group the parameters to mktime together. The more parameters you add, the more accurate your time becomes.

You can compare times using the comparison methods (an earlier time is less than a later time), and if you subtract one time from another, you'll get the number of seconds between them. Play around with it!

A Few Things to Try

One billion seconds... Find out the exact second you were born (if you can). Figure out when you will turn (or perhaps when you did turn?) one billion seconds old. Then go mark your calendar. Happy Birthday! Ask what year a person was born in, then the month, then the day. Figure out how old they are and give them a big SPANK! for each birthday they have had. The Hash Class

Another useful class is the Hash class. Hashes are a lot like arrays: they have a bunch of slots which can point to various objects. However, in an array, the slots are lined up in a row, and each one is numbered (starting from zero). In a hash, the slots aren't in a row (they are just sort of jumbled together), and you can use any object to refer to a slot, not just a number. It's good to use hashes when you have a bunch of things you want to keep track of, but they don't really fit into an ordered list. For example, the colors I use for different parts of the code which created this tutorial:

```
colorArray = []  # same as Array.new
colorHash  = {}  # same as Hash.new

colorArray[0]         = 'red'
colorArray[1]         = 'green'
colorArray[2]         = 'blue'
colorHash['strings']  = 'red'
colorHash['numbers']  = 'green'
colorHash['keywords'] = 'blue'

colorArray.each do |color|
  puts color
end
colorHash.each do |codeType, color|
  puts codeType + ':  ' + color
end
```

```
red
green
```

```
blue
strings:  red
numbers:  green
keywords:  blue
```

If I use an array, I have to remember that slot 0 is for strings, slot 1 is for numbers, etc. But if I use a hash, it's easy! Slot 'strings' holds the color of the strings, of course. Nothing to remember. You might have noticed that when we used each, the objects in the hash didn't come out in the same order we put them in. Arrays are for keeping things in order, not hashes.

Though people usually use strings to name the slots in a hash, you could use any kind of object, even arrays and other hashes (though I can't think of why you would want to do this...):

```
weirdHash = Hash.new

weirdHash[12] = 'monkeys'
weirdHash[[]] = 'emptiness'
weirdHash[Time.new] = 'no time like the present'
```

Hashes and arrays are good for different things; it's up to you to decide which one is best for a particular problem.

Extending Classes

At the end of the last chapter, you wrote a method to give the English phrase for a given integer. It wasn't an integer method, though; it was just a generic "program" method. Wouldn't it be nice if you could write something like 22.to_eng instead of englishNumber 22? Here's how you would do that:

```
class Integer
  def to_eng
    if self == 5
      english = 'five'
    else
      english = 'fifty-eight'
    end

    english
  end
end

# I'd better test on a couple of numbers...
puts 5.to_eng
puts 58.to_eng
```

```
five
```

```
fifty-eight
```

Well, I tested it; it seems to work. 😉

So we defined an integer method by jumping into the Integer class, defining the method there, and jumping back out. Now all integers have this (somewhat incomplete) method. In fact, if you didn't like the way a built-in method like to_s worked, you could just redefine it in much the same way... but I don't recommend it! It's best to leave the old methods alone and to make new ones when you want to do something new.

So... confused yet? Let me go over that last program some more. So far, whenever we executed any code or defined any methods, we did it in the default "program" object. In our last program, we left that object for the first time and went into the class Integer. We defined a method there (which makes it an integer method) and all integers can use it. Inside that method we use self to refer to the object (the integer) using the method.

Creating Classes

We've seen a number of different classes of objects. However, it's easy to come up with kinds of objects that Ruby doesn't have. Luckily, creating a new class is as easy as extending an old one. Let's say we wanted to make some dice in Ruby. Here's how we could make the Die class:

```
class Die

  def roll
    1 + rand(6)
  end

end

# Let's make a couple of dice...
dice = [Die.new, Die.new]

# ...and roll them.
dice.each do |die|
  puts die.roll
end
```

```
6
2
```

(If you skipped the section on random numbers, rand(6) just gives a random number between 0 and 5.)

And that's it! Objects of our very own.

We can define all sorts of methods for our objects... but there's something missing. Working with these objects feels a lot like programming before we learned about variables. Look at our dice, for example. We can roll them, and each time we do they give us a different number. But if we wanted to hang on to that number, we would

have to create a variable to point to the number. It seems like any decent die should be able to have a number, and that rolling the die should change the number. If we keep track of the die, we shouldn't also have to keep track of the number it is showing.

However, if we try to store the number we rolled in a (local) variable in roll, it will be gone as soon as roll is finished. We need to store the number in a different kind of variable:

Instance Variables

Normally when we want to talk about a string, we will just call it a string. However, we could also call it a string object. Sometimes programmers might call it an instance of the class String, but this is just a fancy (and rather long-winded) way of saying string. An instance of a class is just an object of that class.

So instance variables are just an object's variables. A method's local variables last until the method is finished. An object's instance variables, on the other hand, will last as long as the object does. To tell instance variables from local variables, they have @ in front of their names:

```
class Die

  def roll
    @numberShowing = 1 + rand(6)
  end

  def showing
    @numberShowing
  end

end

die = Die.new
die.roll
puts die.showing
puts die.showing
die.roll
puts die.showing
puts die.showing
```

```
4
4
6
6
```

Very nice! So roll rolls the die and showing tells us which number is showing. However, what if we try to look at what's showing before we've rolled the die (before we've set @numberShowing)?

```
class Die
```

```
  def roll
    @numberShowing = 1 + rand(6)
  end

  def showing
    @numberShowing
  end

end

# Since I'm not going to use this die again,
# I don't need to save it in a variable.
puts Die.new.showing
```

Hmmm... well, at least it didn't give us an error. Still, it doesn't really make sense for a die to be "unrolled", or whatever nil is supposed to mean here. It would be nice if we could set up our new die object right when it's created. That's what initialize is for:

```
class Die

  def initialize
    # I'll just roll the die, though we
    # could do something else if we wanted
    # to, like setting the die with 6 showing.
    roll
  end

  def roll
    @numberShowing = 1 + rand(6)
  end

  def showing
    @numberShowing
  end

end

puts Die.new.showing
```

```
1
```

When an object is created, its initialize method (if it has one defined) is always called.

Our dice are just about perfect. The only thing that might be missing is a way to set which side of a die is

showing... why don't you write a cheat method which does just that! Come back when you're done (and when you tested that it worked, of course). Make sure that someone can't set the die to have a 7 showing!

So that's some pretty cool stuff we just covered. It's tricky, though, so let me give another, more interesting example. Let's say we want to make a simple virtual pet, a baby dragon. Like most babies, it should be able to eat, sleep, and poop, which means we will need to be able to feed it, put it to bed, and take it on walks. Internally, our dragon will need to keep track of if it is hungry, tired, or needs to go, but we won't be able to see that when we interact with our dragon, just like you can't ask a human baby, "Are you hungry?". We'll also add a few other fun ways we can interact with our baby dragon, and when he is born we'll give him a name. (Whatever you pass into the new method is passed into the initialize method for you.) Alright, let's give it a shot:

```
class Dragon

  def initialize name
    @name = name
    @asleep = false
    @stuffInBelly     = 10  # He's full.
    @stuffInIntestine =  0  # He doesn't need to go.

    puts @name + ' is born.'
  end

  def feed
    puts 'You feed ' + @name + '.'
    @stuffInBelly = 10
    passageOfTime
  end

  def walk
    puts 'You walk ' + @name + '.'
    @stuffInIntestine = 0
    passageOfTime
  end

  def putToBed
    puts 'You put ' + @name + ' to bed.'
    @asleep = true
    3.times do
      if @asleep
        passageOfTime
      end
      if @asleep
        puts @name + ' snores, filling the room with smoke.'
      end
    end
```

```
  if @asleep
    @asleep = false
    puts @name + ' wakes up slowly.'
  end
end

def toss
  puts 'You toss ' + @name + ' up into the air.'
  puts 'He giggles, which singes your eyebrows.'
  passageOfTime
end

def rock
  puts 'You rock ' + @name + ' gently.'
  @asleep = true
  puts 'He briefly dozes off...'
  passageOfTime
  if @asleep
    @asleep = false
    puts '...but wakes when you stop.'
  end
end

private

# "private" means that the methods defined here are
# methods internal to the object.  (You can feed
# your dragon, but you can't ask him if he's hungry.)

def hungry?
  # Method names can end with "?".
  # Usually, we only do this if the method
  # returns true or false, like this:
  @stuffInBelly <= 2
end

def poopy?
  @stuffInIntestine >= 8
end

def passageOfTime
  if @stuffInBelly > 0
    # Move food from belly to intestine.
    @stuffInBelly     = @stuffInBelly     - 1
    @stuffInIntestine = @stuffInIntestine + 1
```

```
    else  # Our dragon is starving!
      if @asleep
        @asleep = false
        puts 'He wakes up suddenly!'
      end
      puts @name + ' is starving!  In desperation, he ate YOU!'
      exit  # This quits the program.
    end

    if @stuffInIntestine >= 10
      @stuffInIntestine = 0
      puts 'Whoops!  ' + @name + ' had an accident...'
    end

    if hungry?
      if @asleep
        @asleep = false
        puts 'He wakes up suddenly!'
      end
      puts @name + '\'s stomach grumbles...'
    end

    if poopy?
      if @asleep
        @asleep = false
        puts 'He wakes up suddenly!'
      end
      puts @name + ' does the potty dance...'
    end
  end

end

pet = Dragon.new 'Norbert'
pet.feed
pet.toss
pet.walk
pet.putToBed
pet.rock
pet.putToBed
pet.putToBed
pet.putToBed
pet.putToBed
```

```
Norbert is born.
You feed Norbert.
You toss Norbert up into the air.
He giggles, which singes your eyebrows.
You walk Norbert.
You put Norbert to bed.
Norbert snores, filling the room with smoke.
Norbert snores, filling the room with smoke.
Norbert snores, filling the room with smoke.
Norbert wakes up slowly.
You rock Norbert gently.
He briefly dozes off...
...but wakes when you stop.
You put Norbert to bed.
He wakes up suddenly!
Norbert's stomach grumbles...
You put Norbert to bed.
He wakes up suddenly!
Norbert's stomach grumbles...
You put Norbert to bed.
He wakes up suddenly!
Norbert's stomach grumbles...
Norbert does the potty dance...
You put Norbert to bed.
He wakes up suddenly!
Norbert is starving!  In desperation, he ate YOU!
```

Whew! Of course, it would be nicer if this was an interactive program, but you can do that part later. I was just trying to show the parts directly relating to creating a new dragon class.

We saw a few new things in that example. The first is simple: exit terminates the program right then and there. The second is the word private which we stuck right in the middle of our class definition. I could have left it out, but I wanted to enforce the idea of certain methods being things you can do to a dragon, and others which simply happen within the dragon. You can think of these as being "under the hood": unless you are an automobile mechanic, all you really need to know is the gas pedal, the brake pedal, and the steering wheel. A programmer might call those the public interface to your car. How your airbag knows when to deploy, however, is internal to the car; the typical user (driver) doesn't need to know about this.

Actually, for a bit more concrete example along those lines, let's talk about how you might represent a car in a video game (which happens to be my line of work). First, you would want to decide what you want your public interface to look like; in other words, which methods should people be able to call on one of your car objects? Well, they need to be able to push the gas pedal and the brake pedal, but they would also need to be able to specify how hard they are pushing the pedal. (There's a big difference between flooring it and tapping it.) They would also need to be able to steer, and again, they would need to be able to say how hard they are turning the wheel. I suppose you could go further and add a clutch, turn signals, rocket launcher, afterburner, flux capacitor,

etc... it depends on what type of game you are making.

Internal to a car object, though, there would need to be much more going on; other things a car would need are a speed, a direction, and a position (at the most basic). These attributes would be modified by pressing on the gas or brake pedals and turning the wheel, of course, but the user would not be able to set the position directly (which would be like warping). You might also want to keep track of skidding or damage, if you have caught any air, and so on. These would all be internal to your car object.

A Few Things to Try too

Make an OrangeTree class. It should have a height method which returns its height, and a oneYearPasses method, which, when called, ages the tree one year. Each year the tree grows taller (however much you think an orange tree should grow in a year), and after some number of years (again, your call) the tree should die. For the first few years, it should not produce fruit, but after a while it should, and I guess that older trees produce more each year than younger trees... whatever you think makes most sense. And, of course, you should be able to countTheOranges (which returns the number of oranges on the tree), and pickAnOrange (which reduces the @orangeCount by one and returns a string telling you how delicious the orange was, or else it just tells you that there are no more oranges to pick this year). Make sure that any oranges you don't pick one year fall off before the next year.

Write a program so that you can interact with your baby dragon. You should be able to enter commands like feed and walk, and have those methods be called on your dragon. Of course, since what you are inputting are just strings, you will have to have some sort of method dispatch, where your program checks which string was entered, and then calls the appropriate method.

And that's just about all there is to it! But wait a second... I haven't told you about any of those classes for doing things like sending an email, or saving and loading files on your computer, or how to create windows and buttons, or 3D worlds, or anything! Well, there are just so many classes you can use that I can't possibly show you them all; I don't even know what most of them are! What I can tell you is where to find out more about them, so you can learn about the ones you want to program with. Before I send you off, though, there is just one more feature of Ruby you should know about, something most languages don't have, but which I simply could not live without: blocks and procs.