Version 0.5

A Little Ruby, A Lot of Objects

Chapter 1: We've Got Class...

What’s this? 1	The Integer 1
How can I make a 2?	2
What’s another way?	2 × 1 But that seems silly.
Bear with me. How can I compute a 6?	3 × 2 × 1
How about 24?	4 × 3 × 2 × 1
Does all this look familiar?	Yes. Isn’t it a function called factorial?
Right. Do you know what this means when you see it in a math textbook? 5!	It means "5 factorial". It computes the value 120 like this: 5 × 4 × 3 × 2 × 1

Exactly. What do you suppose this Ruby function does?
def factorial(n)
if n == 1
n
else
*n * factorial(n-1)*
end
end
ch1-factorial.rb
	The name tells me it computes factorial, but I’m not sure how.

Let’s figure it out. Can you turn the computation of *n!* into a single multiplication?	If I knew *(n-1)!*, then *n!* would be *n* × *(n-1)!*
Look familiar?	Yes, that’s like this line of the *def*: *n * factorial(n-1)*

But what happens if the *n* in *n!* is 1?	I better not multiply by zero, so I suppose I should stop.
Stop?	I mean I shouldn’t multiply the argument 1 by anything. I know the answer is 1 without multiplying.
Do you see that in the definition of *factorial*?	Yes. That looks like the *if* statement that returns *n*: *if n == 1* *n*
So can you describe *factorial* in words?	"If the argument *n* is 1, the result is 1. Otherwise, the result is *n * factorial(n-1)*."
And what is the result of this? *factorial(5)*	120, because that’s the result of *5 * factorial(4)*, which is in turn *4 * factorial(3)*, which is *3 * factorial(2)*, which is *2 * factorial(1)*, which is *1*.
This is an interesting style of programming – breaking problems into smaller pieces, all solved in the same way. Would you like to know more about it?	I would.
The book to read is *The Little Schemer*, by Daniel P. Friedman and Matthias Felleisen.	OK. But why should I keep reading this book?
You already bought it.	Actually, I’m just browsing in the bookstore. I happened to pass it while I was jogging vigorously and healthily after consuming a breakfast of cauliflower and wheat germ.
Oh. Well, this book is about a different thing. It’s about object-oriented programming in its most free and most fundamental form.	That sounds interesting, but I have no idea what an "object" is.
What if I told you this was an object: *1*	I would be unimpressed. What does that mean?
It means that you can do more to it than multiply and divide.	Such as?

What do you suppose this means? *1.next*	*2*?
Right. Can you describe what’s going on?	The Integer object *1* is asked for the next Integer, which is *2*.
The jargon is that *1* is sent the *next* **message**, and it **answers** (or **returns**) *2*. And what does this mean? *1.next.next*	*3*, because *1.next* is *2* and *2.next* is *3*. But somehow this doesn’t seem an improvement on 1 + 1 + 1.
It isn’t – yet. But what do you suppose this would mean? *5.new_factorial*	Perhaps it would compute *5!* in a new way, a way with messages. It would send the *new_factorial* message to *5*, which would answer the result *120*. But would that work if I tried it?

Not yet. First we have to tell the Integers how *new_factorial* works. That means defining a **method**. A method is the function that’s invoked when a message is received by an object.
 	
We'll define Integer's *new_factorial* like this:
 	
class Integer
  def new_factorial
    ???
  end
end
 	
How do you think *new_factorial* should work?
	Roughly like *factorial* does.
	5.new_factorial should multiply *5* by *4.new_factorial*.

Using the structure of *factorial* for *new_factorial*, we get this:

```
class Integer
  def new_factorial
    if ??? == 1
      ???
    else
       ??? * (??? - 1).new_factorial
    end
  end
end
```

Why are the **???** marks there?

factorial took an argument *n*, which was used in those places. *new_factorial* doesn't have an argument. It doesn't need one. The number to compute with is the Integer *new_factorial* is sent to.

So we need something other than *n* to use in those spots.

Within the definition of any method, *self* always means the object itself.

So here is *new_factorial*:

```
class Integer
  def new_factorial
    if self == 1
       self
    else
       self * (self - 1).new_factorial
    end
  end
end
```

ch1-new-factorial.rb

Can you say that in words?

"I am an Integer.

To compute *new_factorial*, I first check whether I am *1*. If so, I return myself, *1*, the factorial of *1*.

If I'm bigger than *1*, the right result is obtained by multiplying me by the factorial of the number one less than me."

Excellent. And what does *5.new_factorial* do?

5.new_factorial sends the message "*new_factorial*" to an object of class Integer, which responds by invoking the method of the same name and returning its result. Is that right?

Exactly.	That’s pretty neat. I confess that I’m a bit disappointed, though, that there are two kinds of computation: message sends like *new_factorial*, and ordinary multiplications.
Ah, but there really aren’t. Let’s be explicit. What do you suppose is the result of this? *5.send("new_factorial")*	That seems to be another way of writing "send the message *new_factorial* to *5*", so I suppose the answer is *120*.
Precisely. And what do you suppose is the result of this? *3.send("*", 2)*	Send the message * to the object *3*, giving it the argument *2*? That would mean the same thing as this: *3 * 2* That is, *6*
Right again. What happens in response to *3*2* is the same old (or, rather, new) message sending. "*3 * 2*" is just *syntactic sugar*.	Agh! Sugar is poison!
The designers of some languages agree. They use less syntactic sugar. Everything is more explicitly a message send. But people have grown up expecting some things, like arithmetic, to look a certain way, so Ruby follows that convention.	But underneath, all computation consists of sending messages to objects, possibly including other objects as arguments. When I write a program, I’ll be continually saying, "O object, please do such-and-so for me, using these other objects to help", right?
Exactly. In some cases, you’ll be thinking explicitly in those terms. In others, you’ll probably let the syntactic sugar hide the underpinnings from you. You saw another example of syntactic sugar earlier. Where’s the sugar in this? *factorial(5)*	*factorial* is the message, but it doesn’t seem to be sent to any object, unlike *new_factorial*. There must be an implicit receiver when none is explicitly mentioned.

That implicit receiver is *self*. So this:
factorial(5)
is exactly the same as this:
self.factorial(5)

I understand what *self* is when I write something like this:
5.new_factorial
But what is it when I write:
self.factorial(5)
outside of any *class* or *def*?

For the moment, I shouldn't say. But as long as *factorial* doesn't use *self* (which it doesn't), what exactly *self* is doesn't matter. I promise that you'll understand the answer by the end of the book.

Perhaps now would be a good time for a pizza break?

Thanks, but heavy food makes me sleepy. A brisk set of jumping jacks should do the trick.

The First Message

Computation is sending messages to objects.

What's this?
"Ruby"

It's a String.

And this?
"a"

Another String. This one's only one character long.

And this?
"3"

A one-character String, where the one character happens to be 3.

Is *"3"* the same thing as *3*?

No. One's an Integer and one's a String.

What do you suppose this does?
"a".next

It asks for the next string after *"a"*. *"b"* seems like it might be a useful answer.

And how about this?
"aaa".next

"aab"?

Right. What if you sent the *"*"* message to a string, as is done here:
*"Ruby" * 3*
or here:
"Ruby".send("", 3)*

I suppose you'd get *"Ruby"* three times, like this:
"RubyRubyRuby"

Do you think that every message you can send to a String can also be sent to an Integer?

That doesn’t seem sensible. There must be things you can do to Strings that make no sense for Integers.

How about "upper case yourself"?

That doesn’t seem to make sense for Integers.

What’s the result of this?

```
"Ruby".upcase
```

"RUBY"

And the result of this?

```
3.upcase
```

A message about "undefined method 'upcase'".

Can you think of a message to an Integer that wouldn’t make sense for a String?

How about *"Ruby".new_factorial*? That shouldn’t work, because we defined *new_factorial* for Integers.

Integer and String are both **classes**. Judging from what you've seen so far, what are classes for?

An object's class determines which messages it responds to.

If you could look at String’s definition of the method *next*, do you suppose it would look the same as Integer’s definition of *next*?

It doesn’t seem like it could. They behave differently. For example, *"z".next* is *"aa"*. Computing that seems different than computing that *9.next* is *10*.

So two messages can be the same, but that doesn’t mean the methods invoked when they’re sent are. We say that message names are **polymorphic**.

I see, though fancy words like "polymorphic" make me want to jump up and run around in tight little circles.

We won’t use the word much, but the idea is important.

I’m afraid that I don’t see what the big deal is.

Let’s look at a more substantial example. What should be the result of executing this?

```
ascending?(1, 2, 3)
```

true, I suppose, since 3 is bigger than 2 and 2 is bigger than 1.

Can you write *ascending*?

Sure:

```
def ascending?(first, second, third)
  first < second && second < third
end
```

ch1-ascending.rb

What should be the result of executing this? *ascending?("first", "second", "third")*	*true* as well. "third" comes after "second" in the dictionary, and "second" comes after "first".
Will the *ascending?* you wrote work for Strings?	Yes, because it’s not dependent on the classes of its arguments.
Can you be more specific?	*first* < *second* means "send the < message to *first*, passing *second* as an argument". If *first* is an Integer, < means what it normally means for numbers. But if it’s a String, a completely different method is used, one that compares strings in dictionary order.
Have we seen something useful?	It’s nice that I can write one method that works for two classes. Without polymorphism, I’d have to decide whether I wanted to go to the trouble of writing an *ascending?* for Strings.
You’ve seen two classes: Integer and String. You’ll soon see how to create your own classes. When you create your first one, will *ascending?* work with it?	Yes, provided it defines the method <. Shall we do that? I’m eager.
In a moment. I’m feeling a bit peckish right now.	Have a celery stick.

The Second Message

Message names describe the desired result, independently of the object that provides it.

What’s this? *""*	It's a String containing no characters.
And this? *"n"*	A String containing one character.
And this? *"nn"*	A String containing two characters.

How can a String represent an Integer?	A String with *n* characters represents the Integer *n*.
Let's make a class that represents Integers that way. What would be a good name?	How about FunnyNumber?

OK. How would we begin to define FunnyNumber?	*class FunnyNumber*
	...
	end

Suppose I want to create a new FunnyNumber that represents the number 3. How should I do that?	There are three key words in your sentence: "FunnyNumber", "new", and "3". But I'm not sure how to put them together.
What is all computation?	"All computation is sending messages to objects, possibly including other objects as arguments." Just as I can send the *"*"* message to the Integer 3, asking it to multiply itself by 2, perhaps I can send the *"new"* message to the class FunnyNumber, asking it to give me a new FunnyNumber that represents 3.
What would that look like?	*FunnyNumber.new(3)*
Exactly.	There's something odd here, something tantalizing, something invigorating, something that makes me feel able to bench press 150 kilos!
And what's that?	Let me see if I can express it. Up to now, I thought there were two things: objects, and their classes. You sent messages to objects; the object's class determined what methods were invoked. But now, it seems that classes are somehow *themselves* objects that can be sent messages, like *new*. For no reason I can articulate, that just seems incredibly powerful.

It is indeed. Classes as objects are the computational equivalent of performance enhancing drugs. They give you the intellectual leverage to perform great feats of mental strength.

I'm ready! Load up the conceptual barbell!

However, as with physical weights, it's best to build up gradually to the desired goal.

Rats. By the way, to be consistent, you should from now on use the same font for class names as you do for other objects.

You're right. Once we have a *FunnyNumber* class, what would this code do?

```
FunnyNumber.new(3).inspect
```

It would create a new *FunnyNumber*, then send it the *inspect* message. I suppose that puts the *FunnyNumber* into some pleasant format.

Such as *"Funny 3 (nnn)"*, perhaps?

OK. The *"nnn"* is the representation and *3* is the *Integer* represented (because *"nnn"* has length 3).

inspect answers a *String*. To help you define it, let me tell you some of how string formatting works in Ruby. Suppose *s* has the value *"hi"*. This *String*:

```
"s.length = #{s.length}"
```

turns into *"s.length = 2"*. (Still more syntactic sugar.) Anything inside *#{}* is computed, and its value is substituted into the *String* that contains it.

inspect would look something like this:

```
class FunnyNumber
  def inspect
    "Funny #{???.length} (#{???})"
  end
end
```

I'm not sure what the ***???*** is, though, except that it's a *String*. For *FunnyNumber.new(3)*, it's the *String "nnn"*.

Could it be *self*?

I don't think so. *self* is the *FunnyNumber* itself. I'm looking for something that's the *String* that *FunnyNumber* uses to represent *Integers*.

Let's just call it *@rep*, short for "representation". *@rep* will be given a value when the *FunnyNumber* is created (via *FunnyNumber.new*).

So here's *inspect*:

```
class FunnyNumber
   def inspect
     "Funny #{@rep.length} (#{@rep})"
   end
 end
```

To be able to use *inspect*, you need a method defined below. You can read on before trying *inspect*, or you can load ch1-funnynumber.rb now.

Correct.

The @ in *@rep* must mean something. The argument *n* to *factorial* didn't have an @ sign in front of it.

It means that *@rep* is an **instance variable**. When an object is created with *new*, it's called an **instance** of its class. The instance variables are shared by all of that object's methods.

So any method that I write for *FunnyNumber* can use *@rep* when it needs to use the representation.

Yes. Where does *@rep*'s initial value come from?

It must be first created as a result of the call to *new*:

```
      FunnyNumber.new(3)
```

FunnyNumber.new creates a *FunnyNumber* instance. *new* takes the argument *3*, which should be used to initialize *@rep* with the representation for *3* (which is *"nnn"*).

You're implying that one object (the class *FunnyNumber*) should reach into another (the instance it creates) and set its instance variable.

Would that be a problem?

Perhaps not, but it would be annoyingly inconsistent. Before, we concluded that all computation is sending messages to objects, asking them to do something. Here, the *FunnyNumber* class isn't asking, it's ripping open the instance and messing with its guts.

Put so graphically, that does sound unappealing. Perhaps the *FunnyNumber* class, having created the instance, should send it a message called *initialize*.

So *new* would look something like this:

```
def new(an_integer)
  instance = ??? instance creation magic
  instance.initialize(an_integer)
  instance
end
```

What does the *instance* alone on a line mean?

It means that the value of the whole method is the newly-created instance. That's what *new* answers.

That's what *new* should look like. You don't have to write *new*, though, because it's provided automatically by Ruby.

I do have to write *initialize*.

It would look like this:

```
class FunnyNumber
  def initialize(from_integer)
    ???
  end
end
```

What should ***???*** be?

How about this?

```
def initialize(from_integer)
  @rep = "n" * from_integer
end
```

That works because this:

    *"n" * 3*

computes this:

    "nnn".

ch1-funnynumber.rb

So, can you describe what this does?

```
FunnyNumber.new(3).inspect
```

new is a method of the *FunnyNumber* class. It creates a new instance, then calls that instance's *initialize* method, passing the value 3.

initialize sets *@rep*, then returns to *new*. *new* answers (or returns) the newly-created object.

That object is sent the *inspect* message, which answers this string:

    "Funny 3 (nnn)"

Whew! That's quite a workout!

You know everything you need to create new classes. Can you add < to *FunnyNumber?*

The skeleton would look like this:

```
class FunnyNumber
  def <(other)
    ???
  end
end
```

I can think of several ways to fill in the ***???***'s.

What's one way that would *not* work?

@argv.length < other.@argv.length

Why not?

The object getting the < message (*self)* can't reach into the argument (*other*) and peek at its instance variables.

You could make the instance variable available via a method:

```
class FunnyNumber
  def rep
    @rep
  end

  def <(other)
    self.rep < other.rep
  end
end
```

But then anyone who wanted to could look at the internal representation.

As a person, I'm fond of my heart (which has a resting pulse rate of 52 beats per minute, by the way), but I don't wear it on my sleeve. Objects should be similarly restrained.

How about this?

```
class FunnyNumber
  def length
    @rep.length
  end

  def <(other)
    self.length < other.length
  end
end
```

That's a little more modest, but what does the concept "length" have to do with any kind of "number"? Why should it make any more sense to say this:

FunnyNumber.new(3).length

than this:

3.length?

If I'm going to calculate something from *@rep*, I should calculate something useful.

How about this?

```
class FunnyNumber
  def as_integer
    @rep.length
  end

  def <(other)
    self.as_integer < other.as_integer
  end
end
```

ch1-ascending-funnynumber.rb

Yes, it seems generally useful to convert *FunnyNumbers* to *Integers*.

It's interesting that the name is all that changed – it's still *length* underneath. But if I ever decide to use a different representation – something other than a *String* – I will always be able to make *as_integer* work. I might not be able to make *length* work.

Hiding representations behind general-purpose interfaces is good object-oriented design.

Can you now use our old friend *ascending*?

This is *true*:

```
ascending?(FunnyNumber.new(1),
           FunnyNumber.new(2),
           FunnyNumber.new(3))
```

Shall we move to a stair-climbing exercise machine, then make our heartbeats "greater" by "ascending" its stairs? (Ho, ho!)

I'm going to have a pastry.

See you in the next chapter, then.

The Third Message
Classes provide interface and hide representation.