A Little Ruby, A Lot of Objects

Preface

Welcome to my little book. In it, my goal is to teach you a way to think about computation, to show you how far you can take a simple idea: that all computation consists of sending messages to objects. Object-oriented programming is no longer unusual, but taking it to the extreme – making *everything* an object – is still supported by only a few programming languages.

Can I justify this book in practical terms? Will reading it make you a better programmer, even if you never use "call with current continuation" or indulge in "metaclass hackery"? I think it might, but perhaps only if you're the sort of person who would read this sort of book even if it had no practical value.

The real reason for reading this book is that the ideas in it are *neat*. There's an intellectual heritage here, a history of people building idea upon idea. It's an academic heritage, but not in the fussy sense. It's more a joyous heritage of tinkerers, of people buttonholing their friends and saying, "You know, if I take *that* and think about it like *this*, look what I can do!"

Prerequisites

With effort, someone who didn't know programming could read this book. I expect that most readers will know at least one programming language, not necessarily an object-oriented one.

I use a few simple mathematical ideas in some of the examples. The factorial function is the most complex, and I explain a simplified form of it, rather than assume you know what it is. I don't think the book requires any particular mathematical inclination, so don't be scared off at the first sight of *factorial*.

Using the book

This book is written as a dialogue between two people, one who knows objects well, and one who doesn't. The text builds cumulatively. If you don't understand something about one chapter, you'll likely understand the next chapter even less. So I recommend you read slowly. The characters in the book take frequent breaks. I think that's a good idea.

This book uses Ruby, a freely available language developed by Yukihiro Matsumoto, but it is not a book about Ruby. Ruby constructs are introduced gradually, as they're needed, rather than in any systematic order. They're described only enough to allow you to understand code that contains them.

If you want to try variants of the examples, you may need a little more Ruby knowledge. The example files (see below) define new constructs a little more completely. However, even with the examples, this book is not a Ruby tutorial. If you want to use Ruby for general-purpose programming – and you should, since it's a wonderful rapid-development language for many types of applications - the book to read is *Programming Ruby*, by David Thomas and Andrew Hunt (available online at www.rubycentral.com/book/index.html). You'll find that Ruby has many more features than this book describes.

Notation

Ruby text and values printed by the Ruby interpreter are in *italic font*. Everything else is in normal font. Important terms are in **bold** when they're defined.

Sometimes, one participant will show a partially completed snippet of Ruby code. The unfinished part is indicated with ***???***:

```
def finish_this
  ???
end
```

Bold italic font is used to draw your attention to a part of some Ruby code

```
class Something
  def some_function
    "look here"
  end
end
```

Running the examples

I recommend you play with the examples as you read.

As of this writing, Ruby works on Unix and Windows. It is available from www.ruby-lang.org. The Windows download comes from www.rubycentral.com, at www.rubycentral.com/downloads/ruby-install.html.

I recommend you use the Ruby interpreter `irb`. Here's an example:

```
> irb
irb(main):001:0> 1 + 1
2
irb(main):002:0>
```

All of the examples in the book are available from www.visibleworkings.com/little-ruby/source. At the points in the text where an example is complete, a marginal note names the example's file:

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.

You can either cut and paste the example into `irb`, or load the example into Ruby like this:

```
> irb
irb(main):001:0> load 'ch1-factorial.rb'
true
irb(main):002:0>
```

(This assumes you're running `irb` in the directories where the examples live.) Thereafter, you can type things like this:

```
irb(main):002:0> factorial 5
120
irb(main):003:0>
```

Acknowledgements

This book was inspired by *The Little Lisper*, by Daniel P. Friedman and Mattthias Felleisen. I fell in love with their book around 1984. The fourth edition, titled *The Little Schemer*, is still available. If you like this book, you'll like that one too, especially because it treats computation from a different perspective.

The Little Lisper is famous for using food in its examples. As the authors say, "[it] is not a good book to read while dieting." As an ironic homage to my inspiration, one of the characters here is an exercise freak. In that way, this is a very different book.

I received help and encouragement from people who read drafts: Al Chou, Mikkel Damsgaard, Joani DiSilvestro, Pat Eyler, Darrell Ferguson, Tammo Freese, Hal E. Fulton, Ned Konz, Dragos A. Manolescu, Dawn Marick, Pete McBreen, Nat Pryce, Christopher Sawtell, Kevin Smith, Dave Thomas, David Tillman, and Eugene Wallingford.

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.

A Little Ruby, A Lot of Objects

Chapter 2: ...We Get It From Others

Exercise has left a fine sheen of sweat on your brow. Are you ready to descend from the stair-climbing machine?	I am.

Perhaps you should write a method called *descending?*.	I want *descending?(3, 2, 1)* to be *true*:
	
	def descending?(first, second, third)
	  first > second && second > third
	end
	
	ch2-directions.rb

What kinds of classes will *descending?* work with?	Any class that defines >.

Can you write a method *never_descending*? It allows one of the arguments to be equal to the next argument, but not greater.
 	
never_descending?(1, 1, 2) is true
never_descending?(1, 2, 3) is true
never_descending?(2, 3, 2) is false
	def never_descending?(first, second, third)
	  first <= second && second <= third
	end
	
	ch2-directions.rb, again

What kinds of classes will *never_descending?* work with?	Any class that defines <=.
I notice that the sweat on your brow has been joined by a perplexed look.	I'm thinking about how to tell someone else about this suite of methods I'm writing: "*ascending?* works with any class that defines <, *descending?* works with any class that defines >, *never_descending?* works with any class that defines <=..." and so on and on and on for all the methods in the suite.

Those are true statements.

Yes, but who wants to hear all that? What I want to say is more like:

> "You know the normal comparison methods like <? This suite works with any class that implements those."

Or, alternately, "This suite works when the arguments implement the Comparable **protocol**."

I take it that "implements a protocol" is shorthand for "responds to the set of messages named wherever it is that the protocol is defined".

Yes.

Our class *FunnyNumber* doesn't implement the Comparable protocol because it only implements <. For a class to be Comparable, surely it should also implement >.

And so what would happen if you changed the definition of *ascending?* from this:

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

to this:

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

ascending? would stop working with *FunnyNumber*. But it would continue to work with *Integers* and *Strings* because they implement Comparable.

I can see another advantage to protocols. Once I added < to *FunnyNumber*, I was starting down a path – the path to a class whose objects can be compared in a widely accepted way. The Comparable protocol reminds me of everything I need to do to satisfy people's expectations of my code.

Would you like to satisfy those expectations now? You'll need to define <, <=, ==, >=, >, and a method called *between?*.

Heck, no. It would be easy enough to do (once you tell me what *between?* does). For example, I can define > like this:

```
class FunnyNumber
  def >(other)
     self.as_integer > other.as_integer
   end
end
```

But the thought of writing all those trivial methods... well, it doesn't fill me with any great excitement.

Would you be willing to write a single method? It would compare *self* to another object, returning –1 if *self* is less than the other, 0 if it has the same value, and +1 if the other is larger.

Maybe. Is such a method defined for *Integer*?

Yes. Its name is <=> (sometimes called "the spaceship operator").

```
class FunnyNumber
  def <=>(other)
      self.as_integer <=> other.as_integer
   end
end
```

What have I gained?

Can you write comparison methods in terms of <=>?

Sure. For example:

```
class FunnyNumber
  def >(other)
      (self <=> other) == 1
   end
end
```

What have I gained?

If you can do it, so can someone else. And someone else did. They put the Comparable protocol methods in a **module** called *Comparable.* Just as *ascending?* works with any class that responds to <, the *Comparable* module works with any class that responds to <=>.

Show me.

Here's all that *FunnyNumber* needs to do to implement the Comparable protocol:

```
class FunnyNumber
  include Comparable
  def <=>(other)
      self.as_integer <=> other.as_integer
  end
end
```

ch2-comparable-funnynumber.rb

So does this line:

```
include Comparable
```

have the same effect as these?

```
def >(other)
   (self  <=> other) == 1
end
def <(other)
   (self  <=> other) == -1
end
...
```

Almost. There are some differences that we'll learn about later.	Does it have something to do with a module being an object, just like a class is an object?
Indeed it does. Modules and classes are very closely related. Would you have to include *Comparable* in order to say that *FunnyNumber* implements the Comparable protocol?	I suppose if I wanted the extra work, I could implement <, >, and all the other Comparable methods myself.
Implementing a protocol is a matter of which messages a class responds to. Including a module is just a convenient way of implementing a protocol.	So the most important thing about a protocol is that it's an agreement among programmers. It's a way for me to tell my friends what kind of thing my class is.
Would you like to learn another way to add a protocol and the methods that implement it to your class?	Yes. But probably you should first interrupt the conversation with one of your messages.

The Fourth Message

Protocols group messages into coherent sets.

If two different classes implement the same protocol, programs that depend only on that protocol can use them interchangeably.

Suppose we want *FunnyNumber* to ...	I'm getting tired of *FunnyNumber*. Can we have something that has more to do with the real world?
Okay. What's the realest part of the real world?	Exercise.
As you wish. After you finished exercising, I noticed you writing something down in a notebook. What was it?	I record the results of exercising: the number of calories consumed and so forth.
Let's begin, then, by creating a class that models the simplest exercise machine you use. What would that be?	Probably the rowing machine.

So we want a class that represents a single session on a particular rowing machine.

```
class RowingSession
  ...
end
```

How would you identify a session?

By the name of the rowing machine and the amount of time spent on it.

```
class RowingSession
  def initialize(name, time)
    @name = name
    @time = time
  end
end
```

What have you done here?

I've written the *initialize* method that will be called by something like:

```
RowingSession.new("buffy", 30)
```

It assigns the given name and time to instance variables.

"Buffy the rowing machine"?

Look, I don't pick the names, I just use the machines.

How would you print a report on the calories consumed?

(You'll want to use Ruby's *print* method. It prints a string to the output. If the string ends with *\n*, *print* arranges for the next *print* to start on a new line.)

I'd add this method within class *RowingSession*:

```
class RowingSession
  def report
    print "#{@time} minutes on #{@name} = "
    print "#{calories} calories.\n"
  end
end
```

Why did you use two *print* statements to print a single line?

A one-line print statement would be marvelous, but this margin isn't large enough to contain it.

What is *calories*?

It's a method that will compute the number of calories burned from the *@time* spent exercising. I'll also define it within *RowingSession*:

```
class RowingSession
  def calories
    @time * 6
  end
end
```

So how can we use your new class?

```
session = RowingSession.new("buffy", 30)
session.report
```

ch2-rowingsession.rb

And the result is this output:

```
30 minutes on buffy = 180 calories.
```

What's a more complicated exercise machine?

A stair climber. It's computer-controlled, so you can pick more than one type of workout. I use two programs: a steady climb, and one that simulates running hard up a steep hill.

The number of calories you burn also depends on your weight, since you're expending energy lifting yourself.

So you need a new class.

```
class ClimbingSession
  def initialize(name, time, program,
                 weight)
    @name = name
    @time = time
    @program = program
    @weight = weight
  end
end
```

Suppose you'd also written the *calories* method. Could you then use the *report* method you wrote for *RowingSession*?

report is a message you can send to objects of class *RowingSession*. Objects of class *ClimbingSession* wouldn't know anything about it. But I wish I could use it. The code for a *ClimbingSession report* would be identical to *RowingSession*'s version.

Could you use a module to provide *report*?

I could, I suppose. Just as module *Comparable* provides a function < to any class that includes it and defines <=>, I could write a module *CaloryReporter* that provides *report* to any class that includes it and defines *@time*, *@name*, and *calories*.

But, frankly, the connection between the two *Session* classes seems tighter than the connection between *Comparable* and *FunnyNumber*.

It does, doesn't it? For a clue as to the connection, notice the shorthand you used: "the two *Session* classes".

When the differences between a *ClimbingSession* and a *RowingSession* didn't matter, I abbreviated to *Session*. In a sense, I was referring to an imaginary class that captured what was common between the two kinds of sessions.

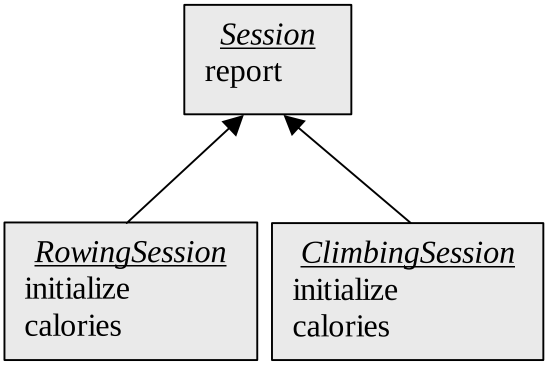
Is method *report* an example of what you want to be common between the two kinds of sessions?

Yes... I want to move *report* into a more "generic" class, because you can report on calories burned for any kind of *Session*.

```
class Session
  def report
     print "#{@time} minutes on #{@name} = "
     print "#{calories} calories.\n"
  end
end
```

If you're trying these examples out in IRB, exit and restart it before defining the above class.

Let's draw a picture of the three classes and where the methods will live.



Now you need a way to say that a *RowingSession* is a kind of Session.

How about this notation?

```
class RowingSession < Session
  def initialize(name, time)
    @name = name
    @time = time
  end

  def calories
    @time * 3
  end
end
```

ch2-rowingsession-as-subclass.rb. If you get a warning message, that means you forgot to exit IRB and restart it.

What does that mean?

A *RowingSession* is a kind of *Session*. Methods specific to *RowingSessions* live in the *RowingSession* class; methods that apply to all *Sessions* live in the *Session* class.

Object-oriented people say that *RowingSession* is a **subclass** of *Session* and (conversely) *Session* is a **superclass** of *RowingSession.*

What is the result of this?

row_sess = RowingSession.new("buffy", 30)

It creates a *RowingSession* object. The arguments to *new* are given to the *initialize* method defined in *RowingSession.*

What is the result of this?

row_sess.report

The *RowingSession* object is sent the *report* message. *RowingSession* doesn't define a *report* method. But, since *RowingSession* is a subclass of *Session*, Ruby looks for *report* there. It finds it and uses it.

More specifically, the result is just as before:

30 minutes on buffy = 180 calories.

We say that *RowingSession* **inherits** *report* from *Session.*

What would *ClimbingSession* look like? (Don't bother completing *calories* yet.)

```
class ClimbingSession < Session
  def initialize(name, time, program,
                    weight)
    @name = name
    @time = time
    @program = program
    @weight = weight
  end

  def calories
    ...
  end
end
```

Notice anything about the two versions of *initialize*? (*RowingSession*'s and *ClimbingSession*'s)

They have two lines in common:

```
@name = name
@time = time
```

Because all *Sessions* will involve a named machine and a time spent on it, I wish I could move those lines into the *Session* class.

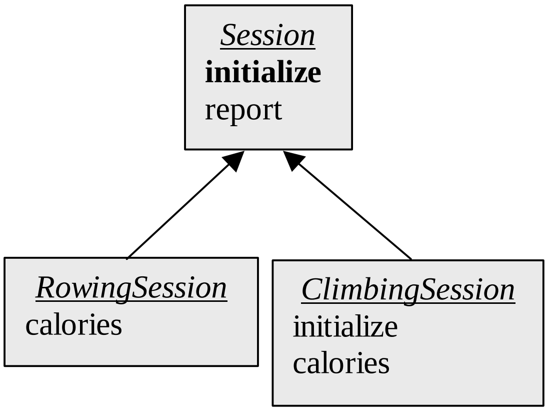
Can you do that for *RowingSession*?

All I need to do is move the definition of *initialize* from *RowingSession* to *Session*:

```
class Session
  def initialize(name, time)
    @name = name
    @time = time
  end
end
```

ch2-rowingsession-initialize.rb

What does our picture look like now?



What will happen as a result of this call?

RowingSession.new("buffy", 30)

The method *new* for the class *RowingSession* will create a *RowingSession* object. Then it will send an *initialize* message to that object. Since *RowingSession* has no *initialize* method, Ruby looks in its superclass, *Session*. It finds it there, so it invokes it.

What about this call, keeping in mind that *ClimbingSession*'s *initialize* hasn't moved?

```
ClimbingSession.new("biff", 23,
                    "hill run",
                    84)
```

You can't run this because *ClimbingSession*'s *calories* hasn't been defined yet.

The method *new* for the class *ClimbingSession* will create a *ClimbingSession* object. Then it will send an *initialize* message to that object. Since *ClimbingSession* defines *initialize*, that one gets invoked. The one in *Session* is ignored.

Can you move the duplicate code from *ClimbingSession* to *Session?*

I'm not sure how. Only two of the lines within *ClimbingSession*'s *initialize* method can be moved. The other two lines have to stay, because they set instance variables unique to *ClimbingSessions*:

```
class ClimbingSession
  def initialize(name, time, program,
                 weight)
    @name = name            # can move
    @time = time            # can move
    @program = program      # must stay
    @weight = weight        # must stay
  end
end
```

What's the problem?

There must be an *initialize* method in *ClimbingSession* to initialize *@program* and *@weight*. Ruby will call that method when it sees

ClimbingSession.new(...)

But how, then, will *Session*'s *initialize* method be called?

Can you show me what you need in the form of code?

I need to know what goes in the **???** slot.

```
class Session
  def initialize(name, time)
    @name = name
    @time = time
  end
end

class ClimbingSession < Session
 def initialize(name, time, program,
                   weight)
   ???
   @program = program
   @weight = weight
 end
end
```

It's something that calls the method of the same name in the superclass.

Call that mechanism *super*.

```
class ClimbingSession < Session
 def initialize(name, time, program,
                   weight)
    super(name, time)
   @program = program
   @weight = weight
 end
end
```

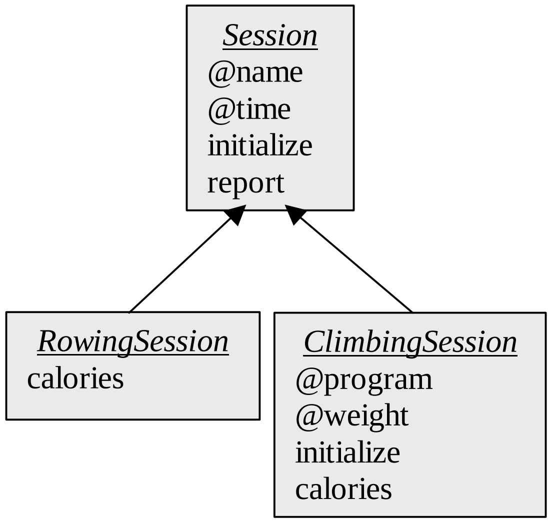
ch2-both-sessions.rb. Exit and reenter IRB before loading it.

Please explain how initialization happens in this case:

```
ClimbingSession.new("biff", 23,
                        "hill run",
                        84)
```

The *new* method on class *ClimbingSession* creates a new object. It sends the *initialize* message to that object, which invokes the *initialize* method from *ClimbingSession*. The first thing that method does is invoke the *initialize* method in the superclass *Session.* After that version of *initialize* initializes *@name* and *@time*, the original *initialize* resumes and initializes *@program* and *@weight*.

Whew! Maybe a picture of the structure, including instance variables, would help.



You've drawn the **inheritance hierarchy** of these classes. *RowingSession* and *ClimbingSession* inherit two instance variables from *Session*. *RowingSession* inherits two methods. *ClimbingSession* inherits only one (*report*), because it **shadows** the other (*initialize*).

This moving of code from place to place – creating superclasses and subclasses as I discover commonality – is exhilarating. But I'm not ashamed to say it also makes me a bit nervous. I'm making the code more pleasing, but what if I break something that used to work?

The technique is called "refactoring". The book to read is Martin Fowler's *Refactoring: Improving the Design of Existing Code*.

I think I'll take a break, run off and buy it.

How about a little summary of inheritance first?

A superclass like *Session* defines protocol for its subclasses. Any class that inherits from *Session* responds to the message *report*. It must implement *calories* for *report* to work, so *calories* is also part of the protocol.

In this way, inheritance is like including a module.

Right. It seems, though, that a module provides implementation (method definitions) for all the messages in its protocol. A class may leave some or all of the implementation to the subclasses. For example, *Session* leaves *calories* to the subclasses.

The Fifth Message
Classes define protocols for their subclasses.

Shall we play class badminton? It will help clarify how inheritance works.

Many people of my culture and with my muscle mass would scorn badminton. But I, being cosmopolitan as well as muscular, realize it is a game of agility, wit, and reflex. So I'm ready.

Here are the rules. In real badminton, two players hit a "shuttlecock" back and forth with rackets. We'll suppose we have two classes, *Super* and *Sub*, instead of rackets. A class "has the shuttlecock" when a method defined in it is executing. It hits the shuttlecock to the other class by causing one of that class's methods to execute.

Oh. Mental agility and wit, not physical. Well, I can do that too.

Serve me up a problem.

Sure.

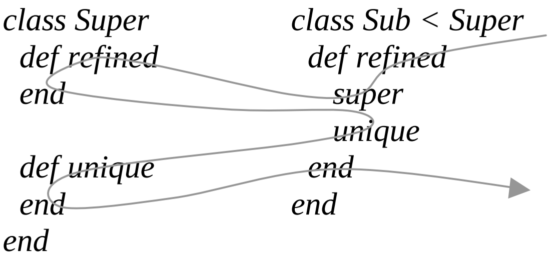
```
class Super                class Sub < Super
  def refined                def refined
  end                          super
                               unique
  def unique                 end
  end                      end
end
```

Given *Sub.new.refined*, what happens?

(If no *initialize* method is defined, all that *new* does is create the object.)

ch2-badminton1.rb

This:



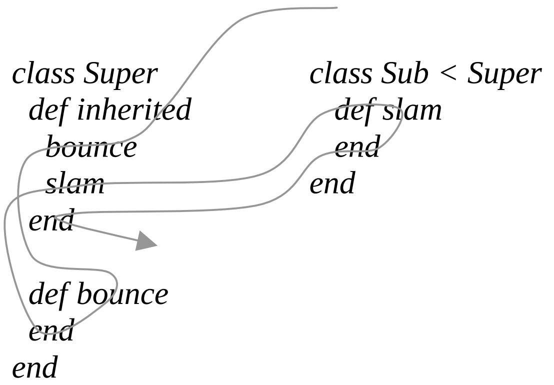
```
class Super                class Sub < Super
  def refined                def refined
  end                          super
                               unique
  def unique                 end
  end                      end
end
```

Sub gets it first, hits it to *Super* (via *super*), who returns it (by returning from *refined*). *Sub* hits it right back by explicitly calling *unique*. *Super* returns it, and *Sub* doesn't hit it back. Point for *Super*.

How about this one?

```
class Super               class Sub < Super
  def inherited              def slam
    bounce                   end
    slam                  end
  end

  def bounce
  end
end
```



```
class Super               class Sub < Super
  def inherited              def slam
    bounce                   end
    slam                  end
  end

  def bounce
  end
end
```

What happens with *Sub.new.inherited*?

An exciting volley! Because *Sub* doesn't define *inherited*, *Super* gets the shuttlecock first. It calls *bounce* – in effect bouncing the shuttlecock up in the air on *Super*'s side of the net. When the shuttlecock comes down (*bounce* returns), *Super slams* it over the net at great speed, expecting *Sub* to be helpless. But *Sub* is ready and returns the volley. *Super*, unprepared for the skillful return, drops the shuttlecock (by returning from *inherited*).

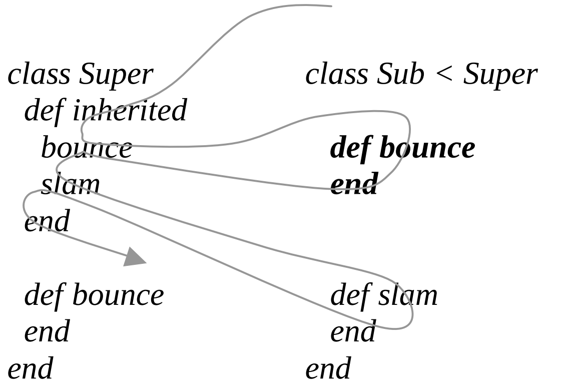
I don't think bouncing the shuttlecock is legal badminton, though.

ch2-badminton2.rb

How about this minor addition?

```
class Super                  class Sub < Super
  def inherited
    bounce                     def bounce
    slam                       end
  end

  def bounce                   def slam
  end                          end
end                          end
```



```
class Super                  class Sub < Super
  def inherited
    bounce                     def bounce
    slam                       end
  end

  def bounce                   def slam
  end                          end
end                          end
```

What happens with *Sub.new.inherited* this time?

Note that *Sub.new* answers a *Sub* object. For a *Sub* object, Ruby will always begin looking for methods in the *Sub* class.

ch2-badminton3.rb

Sub triumphs again! As before, *Super* tried to *bounce* the shuttlecock on its side of the net. This time, though, *Sub* had a *bounce* of its own. Because Ruby will look for methods starting at *Sub*, *Sub*'s *bounce* method was called – converting *Super*'s illegal move into a hit over the net. *Super* – disconcerted – handled *Sub*'s return from *bounce* and tried to *slam* it back. *Sub* returned the slam, and *Super* dropped it. Stellar!

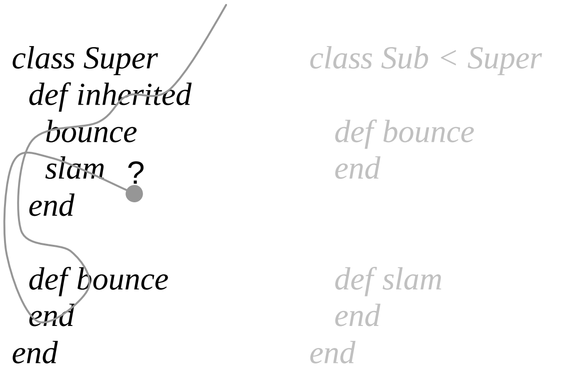
Sub seems to dominate *Super*.

Generally, I find the right side in any sparring, verbal or physical, fares better.

Quite. Let's suppose the classes are as above, but the game begins differently:

Super*.new.inherited*

Since the object created is a *Super*, Ruby will always start looking for methods there. *Sub* is irrelevant. That leads to this:



```
class Super                  class Sub < Super
  def inherited
    bounce                     def bounce
    slam  ?                    end
  end

  def bounce                   def slam
  end                          end
end                          end
```

There is no *slam* method in *Super*, so execution must fail.

Super is what is called an **abstract class**. Abstract classes define protocols. They also provide method implementations and instance variables to the **concrete classes** that inherit from them. But they aren't intended to be **instantiated** (made into instances, created as objects using *new*).

A programmer creating an abstract class should make sure his friends know what methods their subclasses should implement.

And I suppose that suggestive names, like *AbstractSession*, would help avoid mistakes.

Naming is an important issue. Kent Beck's *Smalltalk Best Practice Patterns* is the book to read.

Smalltalk is a different language than Ruby?

Yes, but it is also a "pure" object-oriented language. Most everything you'll see in this book can also be done in Smalltalk.

I'll look it up.

The Sixth Message

If a class and its superclass have methods with the same name, the class's methods take precedence.

We should explore how instance variables work with inheritance. Here's an example:

```
class Super                class Sub < Super
  def super_set(val)         def sub_set(val)
    @val = val                 @val = val
  end                        end

  def super_get              def sub_get
     @val                      @val
  end                        end
end                        end
```

ch2-badminton4.rb

I see two classes. Both of them change variables named *@val*. But is the *@val* in *Super* the same as the *@val* in *Sub*?

Let's see. What is the effect of this?

```
        s = Sub.new
        s.super_set(5)
        s.super_get
        s.sub_get
```

Both *super_get* and *sub_get* answer *5*.

And how about this?

```
s.sub_set("dawn")
s.super_get
s.sub_get
```

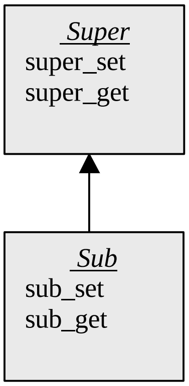
Both *super_get* and *sub_get* answer *"dawn"*.

How do instance variables work with inheritance?

When superclasses and subclasses use the same variable name, they mean the same variable. Variables are not shadowed the way that methods are.

Let's explore why that happens. Please draw *Super* and *Sub*.

Here:



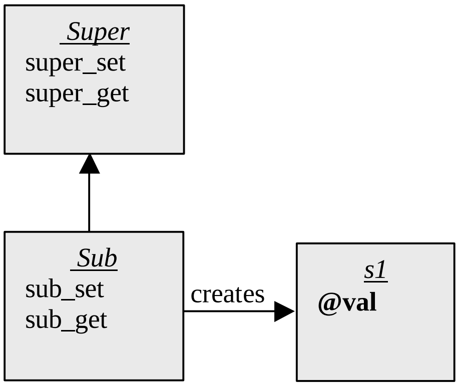
I'm not sure where to put *@val*. It should only go in one place because either class can change it.

Suppose you execute this code:

```
s1 = Sub.new
s1.sub_set(1)
s2 = Sub.new
s2.sub_set(2)
```

Do the two objects have the same value of *@val*?

No. Each instance has a different value. That suggests that an instance should have a separate box, containing its unique instance variables:

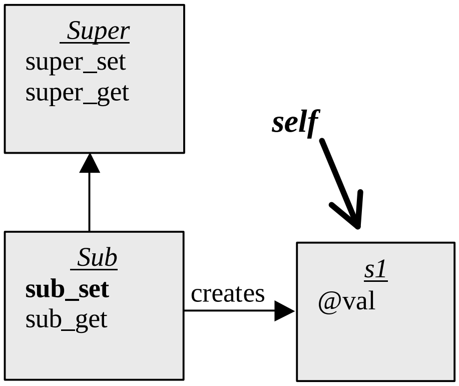


Yes. I earlier had you put instance variables together with methods in one box. That was an oversimplification.

But does this explain why *Super* and *Sub* share the instance variable?

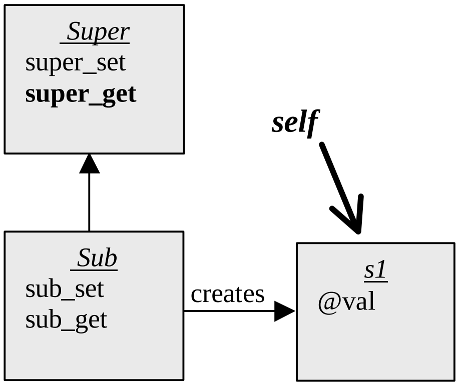
Remember that *self* is always the receiver of a message.

So, given *s1.sub_set(1)*, *self* is *s1*. Here's the picture:



And given *s1.super_get*?

self is the same.



So...?

It's not really that *Super* shares *Sub*'s variable or vice-versa. It's that they both refer to the same variable, stored in *self*.

The Seventh Message

***Instance variables are always found in* self.**

A Little Ruby, A Lot of Objects

Chapter 3: Turtles All The Way Down

You seem a disciplined sort: exercising, eating good food.

If only it were true.

What do you mean?

Sometimes I'm at the store, walking past the ice cream freezer, and I lose all discipline. I reach in and grab some.

A little too much of this, eh?

```
IceCream.new.eat
```

I'm afraid so.

Perhaps we should change the world, once and for all, such that ice cream were not available.

So that *IceCream.new* returned an instance of *Celery*?

We could do that.

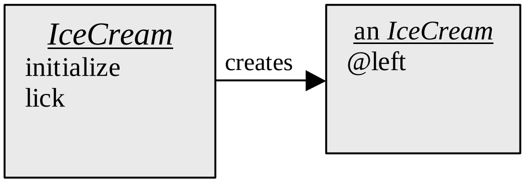
Show me.

We'll work up to it. First, some pictures. Can you describe this class, then draw a picture of it?

```
class IceCream
  def initialize(starting_licks)
    @left = starting_licks
  end

  def lick
    @left = @left – 1
    if @left > 0
      "yum!"
    elsif @left == 0
      "Good to the last lick!"
    else
      "all gone"
    end
  end
end
```

*IceCream initialize*s an *IceCream* instance with the number of times you can lick it. The *lick* method makes the *IceCream* smaller: each time you *lick* it, there's one less lick *@left*. Here are the methods and the instance variable:



Somehow this isn't doing much to wean me from ice cream.

You've shown that *IceCream* creates an instance. Once the instance is created, what is the relationship between it and its class?

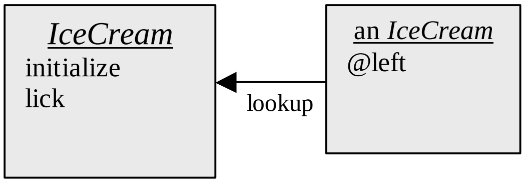
Hint: given this:

anIceCream = IceCream.new(100)

what happens for this?

anIceCream.lick

When an *IceCream* instance receives a message (such as *lick*), it uses the class to find what method implements that message. The arrow below shows that.



I notice that *new* isn't in either box. Where does it belong?

Hmm. It certainly doesn't belong in the instance box on the right. But it shouldn't belong in the class box on the left either.

Why not?

When an *IceCream* instance receives a message, it looks to the left to find the method. If *new* were in the class box, that would mean the instance would respond to *new*, like this:

anIceCream.new(100)

We don't want that.

No, *new* should be something the class responds to, not the instance.

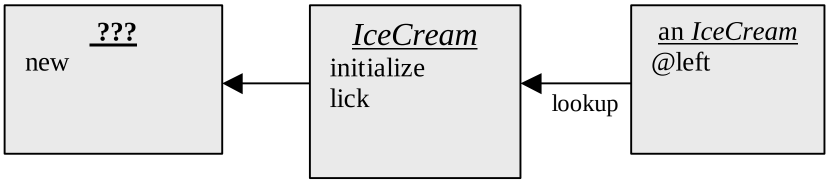
Given this:

IceCream.new(100)

the class is the object that receives the message. So, for consistency, it too should look left to find the right method.

Show me.

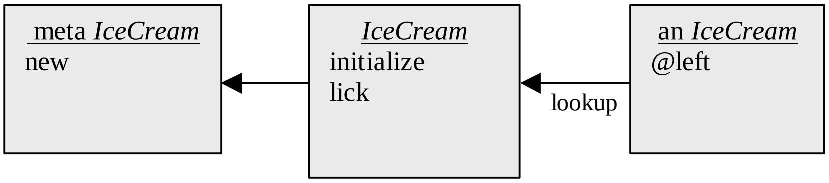
I'll have to borrow some of your space.



I don't know what the name of that leftmost box should be, though.

Such objects are usually called **metaclasses**. "Meta" is supposed to have the connotation of "beside" or "above" or "beyond".

Well, from the perspective of the *IceCream* instance, that new box is beyond the *IceCream* class. So I'll add that name:



All this seems weighty and over-elaborate.

Only because you haven't finished building up your metaclass muscles.

Notice that we initialize our *IceCream* with the number of licks:

```
anIceCream = IceCream.new(100)
```

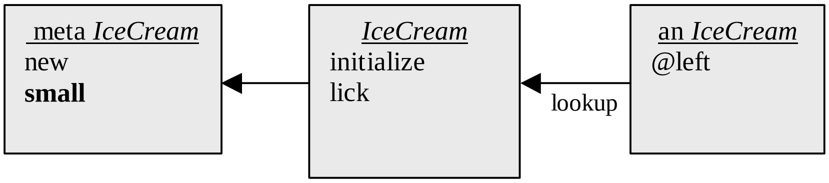
It might be more convenient to create *IceCream* instances in standard sizes.

I myself would choose only a small ice cream.

So add this to the picture:

```
anIceCream = IceCream.small
```

The *small* method goes on the metaclass.



Here's how our new method would be defined:

```
class IceCream
  def IceCream.small
    new(80)
  end
end
```

ch3-small-icecream.rb

I see two odd things about that definition. The first is the name, which is *IceCream.small*. I'm used to method definitions that start like this:

```
class IceCream
  def lick
    ...
```

Prefacing the name of the method with the name of the class tells Ruby that this method applies to the class object itself, not to instances.

FunnyNumber.small is a **class method**. Everything we've defined before now has been an **instance method** (like *lick* or *initialize*).

The format is easy to remember, because you define class methods the same way you use them:

```
def IceCream.small ...

anIceCream = IceCream.small
```

What's the second odd thing?

I am used to typing *IceCream.new*, but the definition of *IceCream.small* refers to an unadorned *new*:

```
def IceCream.small
  new(80)
```

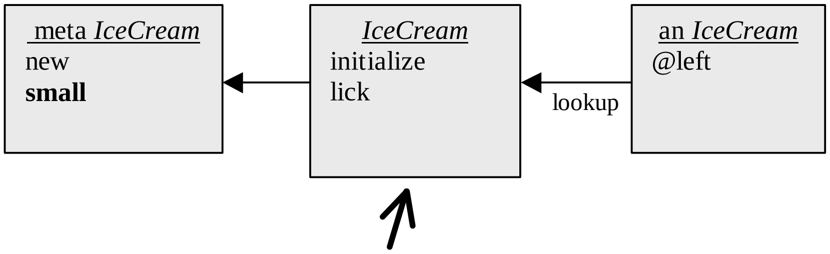
When no object is specified, where is a message sent?

self. So the definition is equivalent to

```
def IceCream.small
  self.new(80)
```

And what object is *self* in that context?

self is always the receiver of the message. This computation started by sending a *small* message to *IceCream*. So *self* can only be the *IceCream* class itself. Like this:



What would be another way of invoking *IceCream.new* within this *def*?

Directly:

```
def IceCream.small
  IceCream.new(80)
```

You now have the tools to change your world. Start a definition of *IceCream.new*.

It's just like any other class method:

```
class IceCream
  def IceCream.new(starting_licks)
    ???
  end
end
```

And what should *IceCream.new* do?

It should make a *Celery*:

```
class IceCream
  def IceCream.new(starting_licks)
    Celery.new
  end
end
```

But how can I be sure it works?

Let's suppose you try to lick the celery.

How perverse!

```
class Celery
  def lick
    "licking celery? yuck!"
  end
end
```

So *IceCream.new(100).lick* should produce *"licking celery? yuck!"*

ch3-icecream-as-celery.rb

And what should *IceCream.small.lick* produce?

The same thing, because *IceCream.small* uses *IceCream.new* (via the implicit *self*).

There's another way to check that you have the right object. All objects in Ruby respond to the *class* message. Try it.

IceCream.small.class answers *Celery*. Say, I notice that *Celery* doesn't have quotes around it, so it's not a *String*.

No, it is the *Celery* class itself.

That means I can send messages to what *class* answers, like this:

```
food = IceCream.small
more_food = food.class.small
```

Both *food* and *more_food* would be instances of *Celery*.

Yes, that's true.	Another example of polymorphism. As long as I know *food* is an instance of a class that obeys the "small portions" protocol, I can create more instances like it. I don't necessarily have to know what kind of food it is.
All class objects obey a protocol: they all implement a *new* method that creates a new instance. Some class methods may extend that protocol to create instances in special ways.	Interesting. Let's have some... celery.

The Eighth Message

Classes are objects with a protocol to create other objects

Did you enjoy your celery?	No. My enthusiasm for eliminating ice cream from the world has vanished.
Perhaps an occasional ice cream wouldn't hurt.	There is something called the "80/20 rule", which advocates having a virtuous diet only 80% of the time.
Let us arrange for you to get ice cream one time out of five.	OK. Then I'll have something to look forward to.
In Ruby, 3*%5* means "what remains after dividing 3 by 5".	In this case, it would be *3*.
And in this case? *13%5*	*3*, again. 13 divided by 5 is 2, with a remainder of 3.
And this? *5%5*	*0*. Ice cream time! I could get celery when the remainder was 1, 2, 3, or 4, then ice cream when it was 0.

Can you sketch what a more palatable *IceCream.new* would look like?

To increment a variable, you can write either this:

variable = variable + 1

or this shorthand:

variable += 1

```
class IceCream
  def IceCream.new(starting_licks)
     ???  += 1
     if ??? % 5 == 0
        IceCream.new(starting_licks)
    else
       Celery.new
    end
  end
end
```

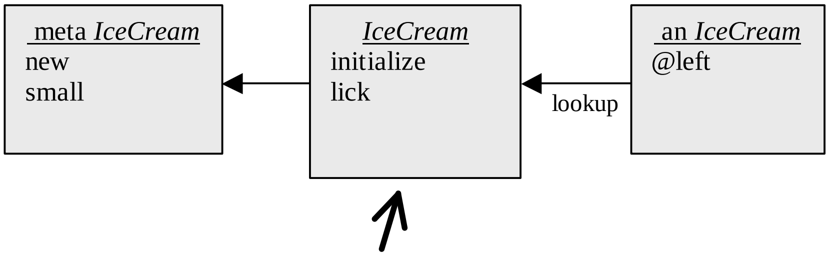
What should I name the variable?

How about *@created*? That's a good name for the number of *IceCream* instances created.

The "@" tells me *@created* is an instance variable. I guess I can use an instance variable in a class, because a class is an object. But I'm not sure how all this will hang together.

Let's use the picture you drew earlier. Within the method *IceCream.new*, what does *self* mean?

self is always the receiver of the message.

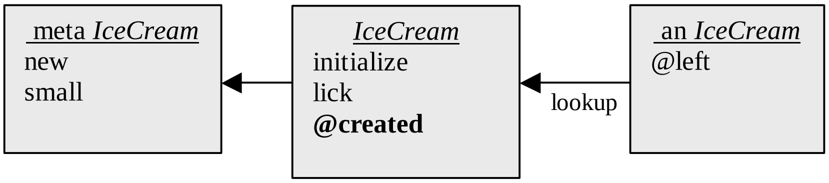


What's the rule for instance variables?

An instance variable's value is always found in *self*.

So when we use an instance variable in a class method, the variable is to be found in ...

... the class! Like this:



So this should work:

```
class IceCream
  def IceCream.new(starting_licks)
     @created += 1
     if @created % 5 == 0
        IceCream.new(starting_licks)
    else
       Celery.new
    end
  end
end
```

Maybe. Is *@created* originally zero?

If an instance variable's value is used before it's ever been set, its value is *nil*.

So the first time *IceCream.new* is called, Ruby will add *1* to *nil*.

Since *nil+1* is nonsense, Ruby will complain of an error.

So I must initialize *@created*. But where?

Anywhere outside an instance method will do.

Right, because initializing *@created* inside an instance method (such as *initialize*) wouldn't refer to the class's *@created* – *self* would be an *IceCream* instance, not *IceCream* itself. How about just sticking it here?

```
class IceCream
  @created = 0
  def IceCream.new(starting_licks)
      ...
  end
end
```

ch3-celery-sometimes.rb

Looks good. Try it out. You can either use something like this:

```
IceCream.new(100).class
```

or this:

```
IceCream.small.class
```

I'll get ice cream on my fifth try. The first *IceCream.small.class* gives me *Celery*. The second, *Celery*. The third, the same. The fourth, the same. The fifth... Hey!

What seems to be the problem?

I got *Celery* again. I am bitterly disappointed.

Can you see why we got *Celery*?

The problem is here:

```
def IceCream.new(starting_licks)
  @created = @created + 1
  if @created % 5 == 0
    IceCream.new(starting_licks)
  else
    Celery.new
  end
end
```

We used *IceCream.new* because that's the way you create an instance. But we're in the middle of redefining *IceCream.new*. So when *@created* is *5*, our new *new* calls itself, which increments *@created* to *6* and so returns a *Celery*.

A problem. We have to do something else.

We have to call the previous version of *new*.

Have we ever done anything like that before?

Yes, sort of. *ClimbingSession* used *super* to call *Session*'s *initialize* method. What would happen if I did the same thing here?

```
def IceCream.new(starting_licks)
  @created = @created + 1
  if @created % 5 == 0
    super(starting_licks)
  else
    Celery.new
  end
end
```

ch3-celery-sometimes-works.rb Exit and restart IRB so that @created is reset to 0.

Try it and see.

Celery. Celery. Celery. Celery. IceCream!

Let's eat.

Wait just one cotton-pickin' minute here. *IceCream* isn't a subclass of anything, so how can it use *super*?

You can find a class's superclass with the *superclass* method.

I use this:

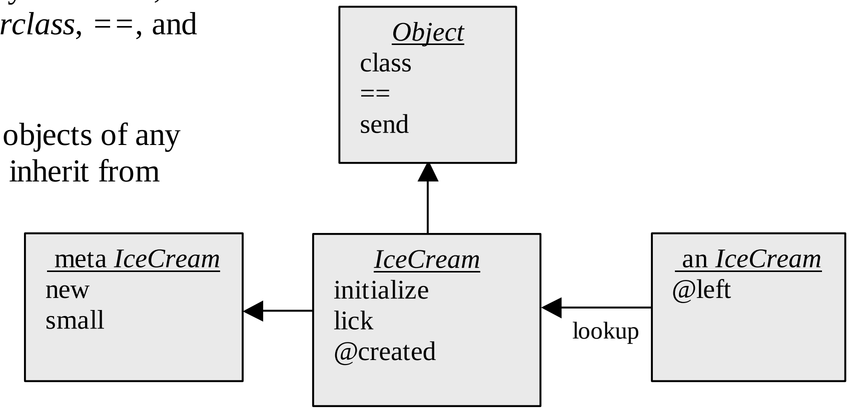
IceCream.superclass

The result is *Object*.

Object is a superclass of all other classes. It defines methods we've been using without thinking about where they're defined, methods like *class*, *superclass*, ==, and *send*.

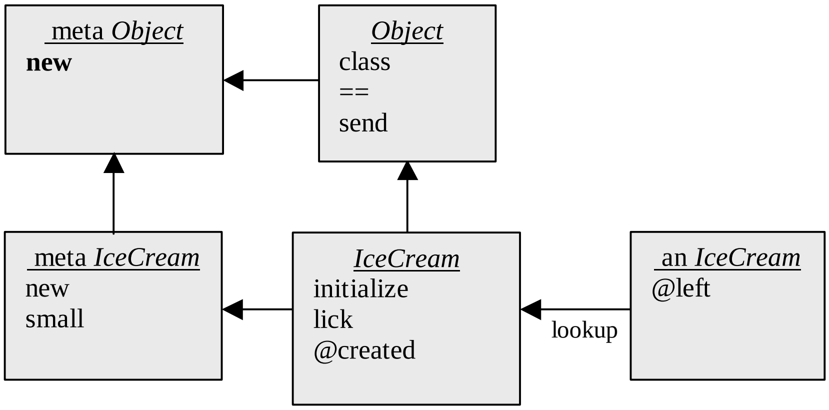
These methods apply to objects of any class, because all classes inherit from *Object.*

That looks like this:



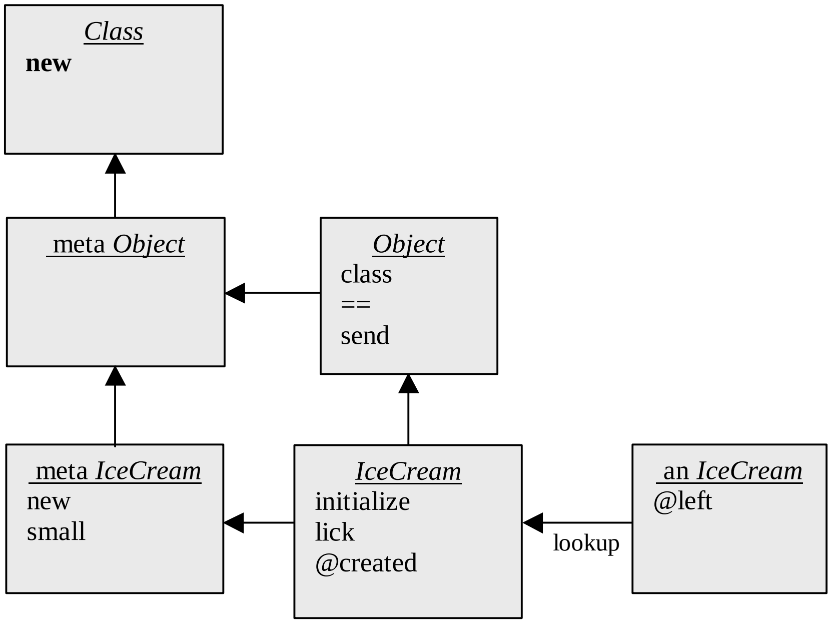
But *new* is not defined in *Object*.

No, otherwise instances could respond to *new* and create new instances. Is *new* defined in a meta *Object*? Like this?



It could be, but for convenience it's defined as an instance method of a class named *Class*. Meta *Object* inherits from it.

Like this:



Now you know what the *super* in *IceCream.new* means.

It means "look above meta *IceCream* for a method *new*". That method is found as an instance method of class *Class*.

Let's review the arrows in this diagram. What does a left pointing arrow mean?

If a message is sent to an object, the left pointing arrow is used to begin the search for a method with the same name.

For example, the *IceCream* class is the place to start searching when an *IceCream* instance is sent the *lick* message.

And meta *IceCream* is the place to start searching when *IceCream* is sent a *new* message.

You can create a generic unadorned *Object* with *Object.new*. Where does the search start in that case?

Meta *Object* is the place to start searching when *Object* is sent a *new* message.

And if no such method is found in the object the arrow points to?	The upward pointing arrow is used to find the next object to check. Because meta *Object* does not define *new*, the search continues in *Class*.
And if no method is found when you hit the topmost object in the column?	The original object does not respond to that message. For example, you may have tried to send *upcase* to an *Integer* or *factorial* to a *String*.
And what is the rule about *self*?	No matter where the method is found, *self* is always the original receiver of the message.
Any questions?	You bet. You said *Class* is a "convenience". Why? And why is it a class instead of a metaclass?
Those are good questions. Let's take a break first. Perhaps sushi is a compromise between the indulgence of ice cream and the ascetic boredom of celery.	Sushi seems oddly appropriate. Let's go!

The Ninth Message
Methods are found by searching through lists of objects.

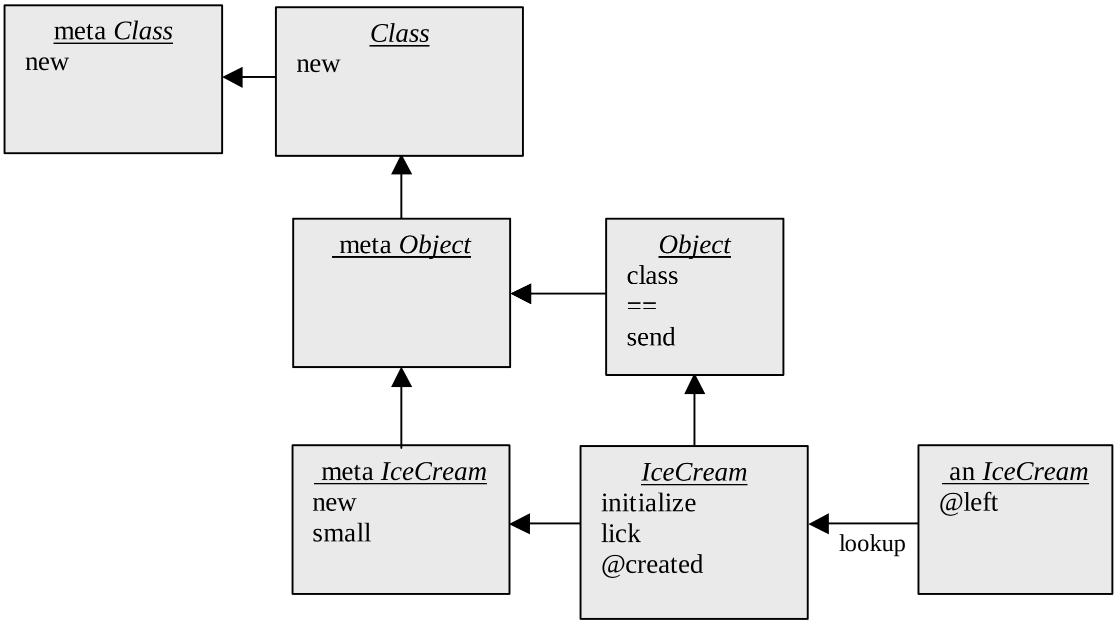
You wanted to know why *Class* is a convenience?	Yes.
What kind of thing is *IceCream.small*?	Because of the tricky code we wrote, most of the time it's a *Celery*. You can find that out like this: *IceCream.small.class*
And what kind of thing is *Celery* itself?	It's a class. You can find that out like this: *Celery.class* The result is *Class*.
Ruby's designer could have eliminated *Class* by putting the *new* method in meta *Object*. Would something like *metaObject* be a better answer for *Celery.class*?	No. *Class* is more suggestive.

Class *Class* is a convenient name to use to suggest behavior common to all classes.	That's true even though, in some sense, the true "class of *Celery*" is meta *Celery*.
Yes. Think of sending the *class* message to an object as a way of getting a hint about what protocol the object obeys.	Just a hint?
Just a hint. We've already seen an example of how the hint can be wrong. *IceCream.class* is a *Class*. Because of that, we expect that *IceCream.new* will produce a new instance of *IceCream*. But it doesn't, not always. We'll later see other ways in which the *class* hint can be wrong.	OK. I accept that *Class* is a convenience and that the *class* method is just a hint.
There's another reason for the *Class* object. What does *Celery.new* do?	It creates a new instance of *Celery*.
How does it do it?	It looks for *new* in *Celery*'s metaclass, eventually finding it in *Class*.
That's how instances are created. How are classes themselves created?	Hmm. *Class.new* seems like a good message.

Yes. Here's a way to create a subclass of *Celery*:
OrganicCelery = Class.new(Celery)
	I was used to this:
	class OrganicCelery < Celery
	end
	But now I see that's syntactic sugar again. Interesting.

We'll see more about that in later chapters. In the meantime, where can this new *new* method be found?

Well, the rule is always to look left, where you find... the meta *Class*. Like this:



Is this too complicated?

All the boxes make it seem complicated, but I guess it's really not. There's a simple rule: you always find methods by starting at an object, calling it *self*, looking left, then looking up. It doesn't matter whether the object is an instance, a class, or your Aunt Marge.

Are you content now?

Except for the fact that our *IceCream* class doesn't work.

What!

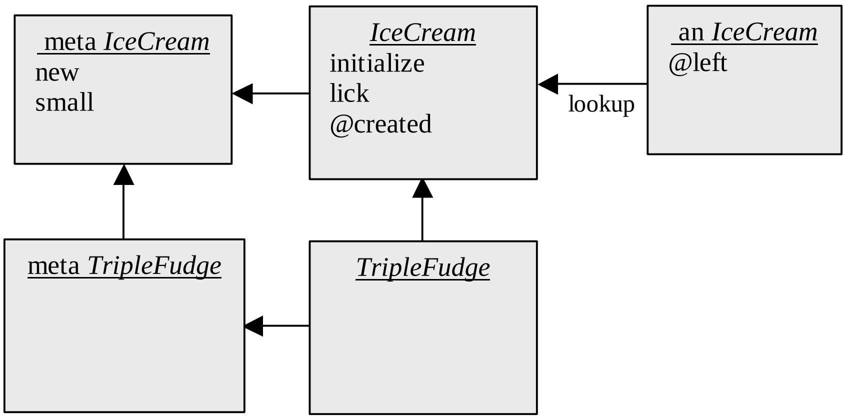
What happens when you do this?

```
class TripleFudge < IceCream
end

TripleFudge.new(1000)
```

Hmm... "undefined method + for nil". I'm perplexed.

A picture will help you understand. Here's the new class:



When *TripleFudge* receives the *new* message, it finds the *new* method in meta *IceCream*.

When that method operates on *@created*, it looks for the variable in *self*.

self is the original receiver of the message: *TripleFudge*...

... which does not contain a variable *@created*.

Actually, it soon does. Ruby executes this line of code inside *IceCream.new*:

@created = @created + 1

That means looking for *@created*'s value inside *self* (*TripleFudge*). When Ruby discovers that the variable does not exist, it creates it.

So *TripleFudge* does have a *@created*, but it's a completely different variable than *IceCream's*. They have the same name, but there's no reason for them to have the same value.

And, since *TripleFudge's* new variable *@created* has never been set, its initial value is...

... *nil*. And the attempt to increment *self* by *1* means sending the message + to *nil*, which is nonsense.

Hence the error message.

It seems confusing for Ruby to create a variable with value *nil* when a program uses a variable that does not exist.

It's really no more confusing than a "variable does not exist" message, once you've seen it a few times. And some programs can usefully take advantage of this behavior.

I'll take your word on that – for now. We need a way to have *IceCream.new* operate on *IceCream's @created* no matter what the original receiver. That's a puzzler.

Hmm... I've got it! To manipulate *IceCream*'s *@created*, we must be inside a method that has *self* set to *IceCream*.

Yes, but *self* is set to *TripleFudge* when we're inside *new*.

So *new* should send a message explicitly to *IceCream*. Within that method, *self* will be *IceCream*.

Such a method could be called *IceCream.allowed?* It says whether to create a *Celery* or an *IceCream*.

```
def IceCream.new(starting_licks)
  if IceCream.allowed?
     super(starting_licks)
  else
    Celery.new
  end
end
```

Write *IceCream.allowed?*, please.

I pull out some of the code that was in our previous version of *IceCream.new*:

```
class IceCream
  def IceCream.allowed?
    @created += 1
    @created % 5 == 0
  end
end
```

ch3-celery-final.rb Exit and restart IRB so that @created is reset to 0

Try it.

I'll mix up requests for plain *IceCream* and for the really good stuff.

IceCream.new(1).class is *Celery*.
TripleFudge.new(99).class is *Celery*.
IceCream.new(1).class is *Celery*.
TripleFudge.new(99).class is *Celery*.
TripleFudge.new(99).class is ***TripleFudge***.
Yes!

Will *TripleFudge.small* work?

Yes. Sending *small* to *TripleFudge* runs this method:

```
class IceCream
  def IceCream.small
    new(80)
  end
end
```

new(80) means *self.new(80)*. So the receiver of *new* will be the same as the receiver of *small* – that is, *TripleFudge.*

So let me ask again: Is this too complicated?

Well, the underlying rules are simple. Look left, then up. *self* is the original receiver. But it can be twisty to keep track of what's where.

That's because we're writing tricky methods that do unusual things. In most cases, you don't have to think about what *self* is or where methods are found.

This is tricky. But whatever doesn't kill me makes me stronger. Nietzsche.

Gesundheit. The fascinating thing about computation is how much you can accomplish with combinations of simple rules.

I'm starting to see that. Tricks like an *IceCream.new* that answers a *Celery*... those can't be anticipated.

A language that provides lots of features will always be missing that one feature you need.

But a language that chooses the right simple rules for you to combine lets you build the features you need.

And it can come with lots of features, too. The book to read about Ruby's features is *Programming Ruby,* by David Thomas and Andrew Hunt.

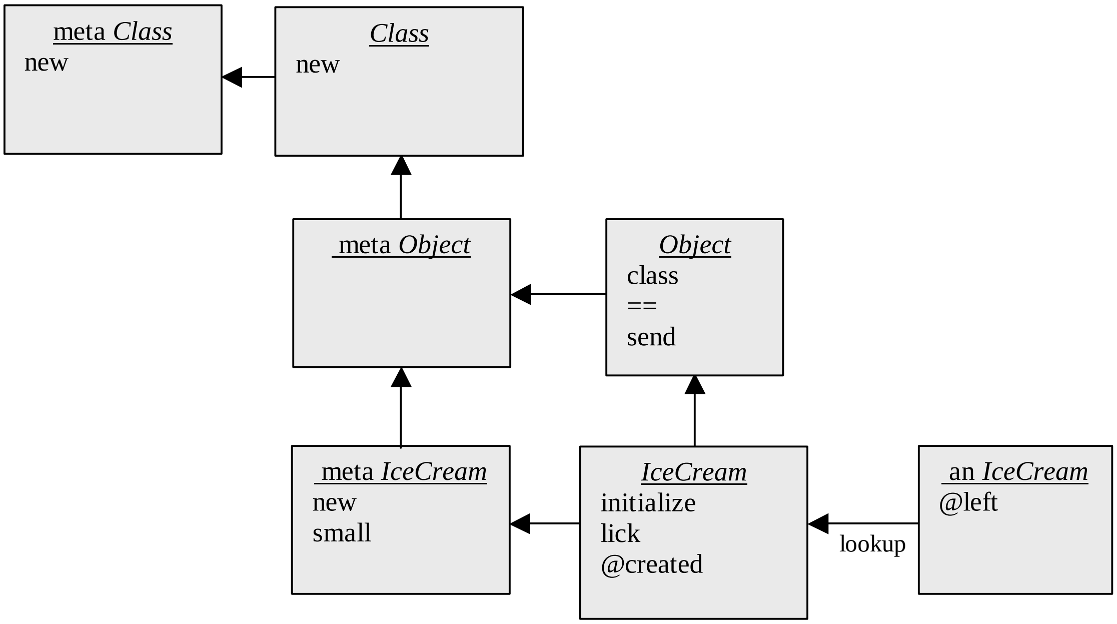
In order to get strong enough to carry all these books you're having me buy, I'm going to have to go the gym and lift some more weights.

The Tenth Message

In computation, simple rules combine to allow complex possibilities

Let's tie up a couple of loose ends. Here is our class picture again.

It's quite familiar now.



What's the answer if you send the *class* message to the *IceCream* instance in the picture?

IceCream.

How is it gotten?

By looking left, then up, from the instance, and finding the *class* method in *Object*. That method answers *IceCream*.

What is the result of *IceCream.class*?

Class, which is appropriate.

How is that result obtained?

You look left and then up, starting at *IceCream*.

And where do you find *class*?

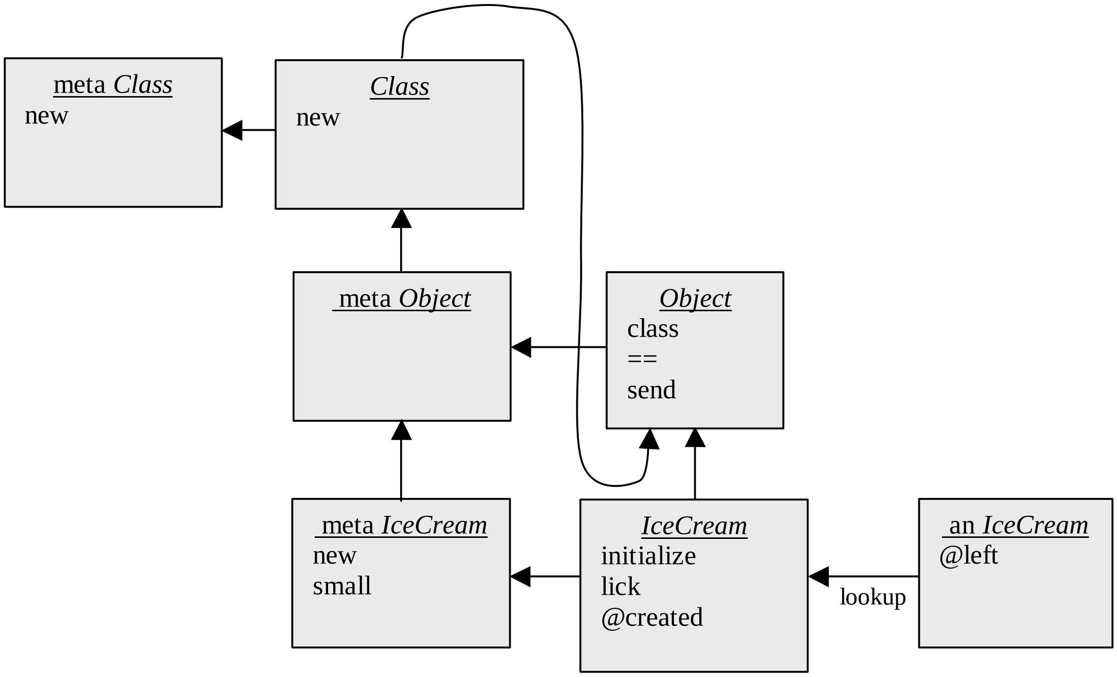
You don't, not in this picture.

Where should you find it?

Object. That means that looking up from *Class* should land you in *Object*.

So the arrow up from *Class* should curve back down to *Object*. Don't fix the picture yet.

I want to. I'd rather have clarity than save paper.



Should there be an arrow up out of meta *Class*?

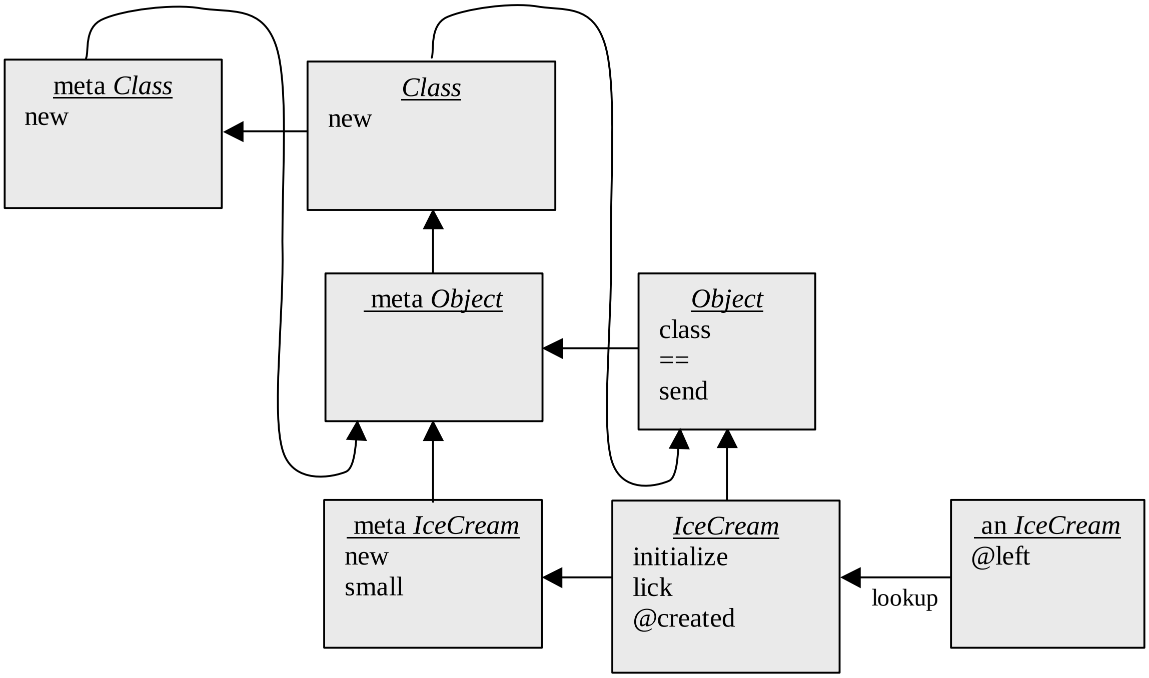
Yes. Since *Class* inherits from *Object*, meta *Class* should inherit from meta *Object*.

Why's that?

Consistency. *Class* has the same relationship to *Object* as *IceCream* does. So meta *Class* should have the same relationship to meta *Object* as meta *IceCream* does.

Now you may draw a picture.

You're very gracious.



So what happens when we send the *class* message to *Class*?

The *class* method is found by looking left and up from *Class*.

And where is it found?

In *Object*. Meta *Class* inherits from meta *Object*, and meta *Object* inherits from *Class*, and *Class* inherits from *Object*.

And what does *Class.class* answer?

Class, like *IceCream*, is a *Class*. That makes sense, because it follows the *new* protocol.

Have we drawn a pretty picture in this chapter?

Nearly as pretty as a picture of an ice cream cone in the window of an ice cream shop. Let's go.

Shall we walk to an ice cream shop?

I know one quite nearby.

The Eleventh Message
***Everything inherits from* Object.**