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.

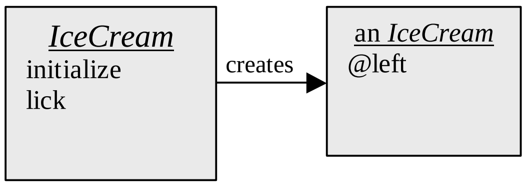
Show me.

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?

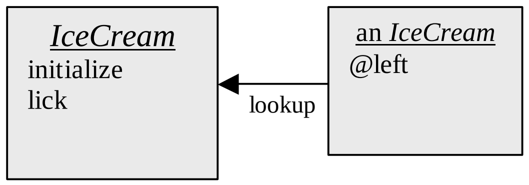
Hint: given this:

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.

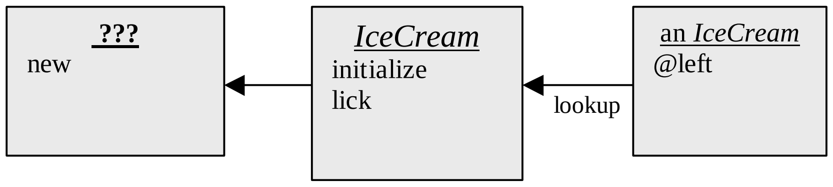
Given this:

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.

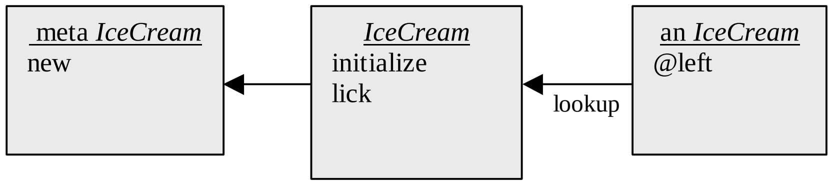
I'll have to borrow some of your space.



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)
```

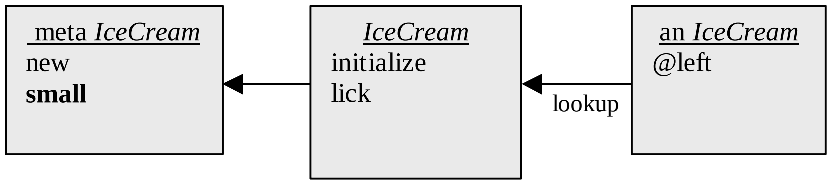
It might be more convenient to create *IceCream* instances in standard sizes.

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)
```

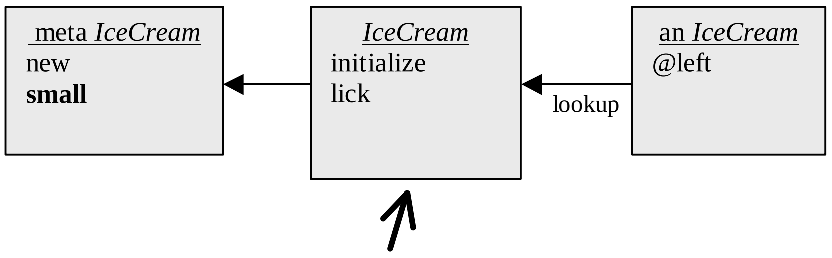
When no object is specified, where is a message sent?

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
```

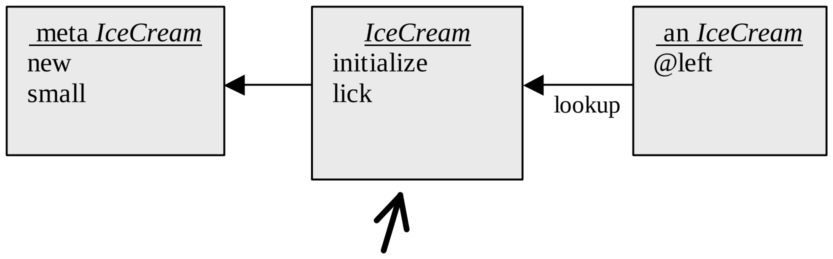
What should I name the variable?

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.

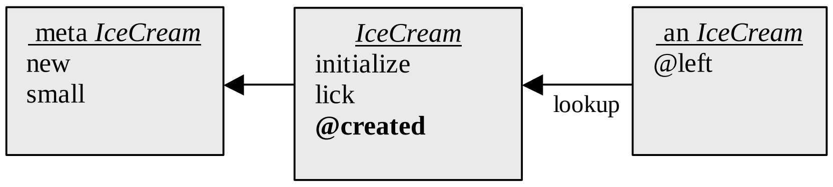


What's the rule for instance variables?

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:

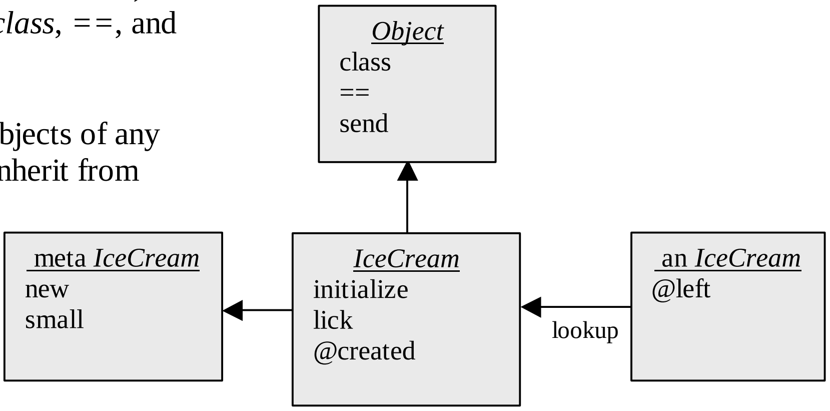
IceCream.superclass

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*.

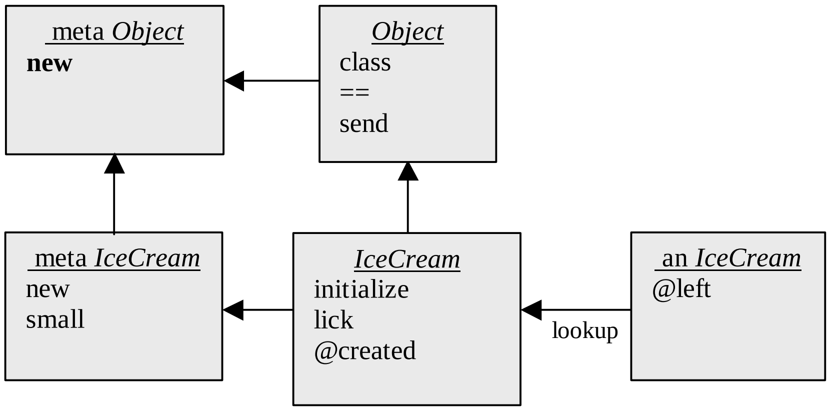
These methods apply to objects of any class, because all classes inherit from *Object.*

That looks like this:



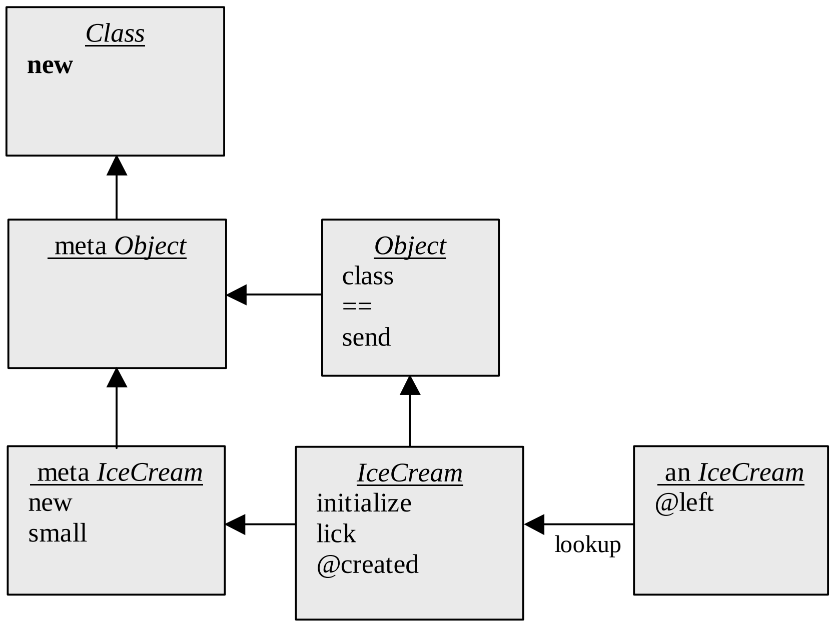
But *new* is not defined in *Object*.

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.

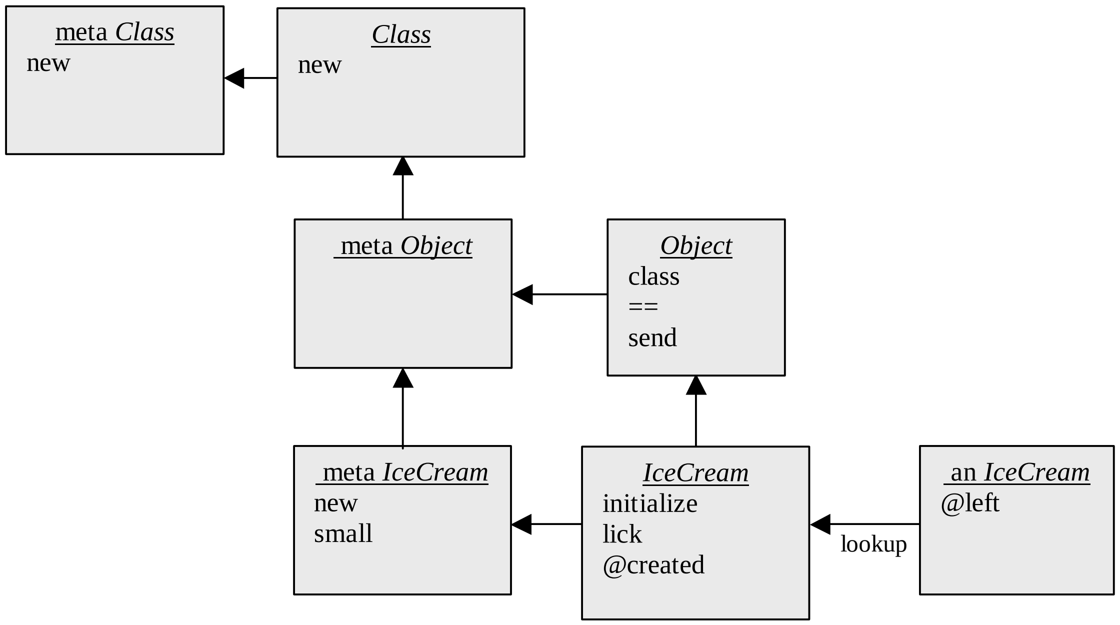
You wanted to know why *Class* is a convenience?	Yes.
What kind of thing is *IceCream.small*?	Because of the tricky code we wrote, most of the time it's a *Celery*. You can find that out like this: *IceCream.small.class*
And what kind of thing is *Celery* itself?	It's a class. You can find that out like 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!

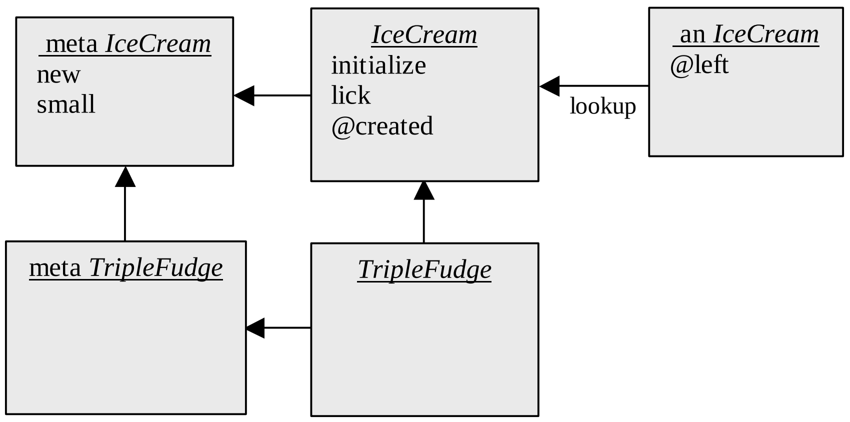
What happens when you do this?

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.

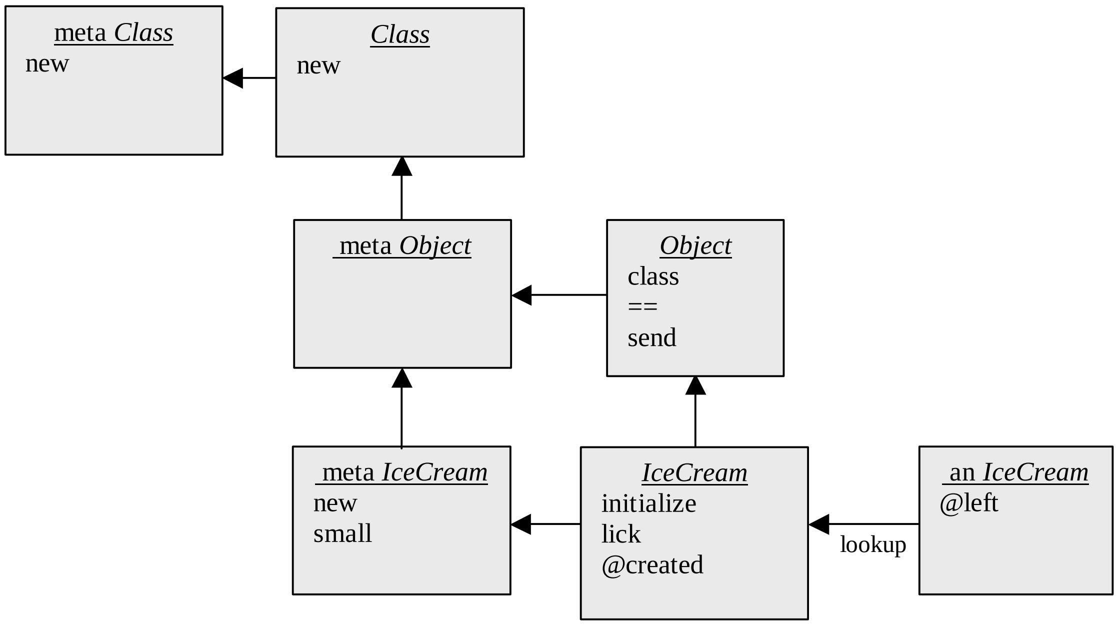
IceCream.new(1).class is *Celery*.
TripleFudge.new(99).class is *Celery*.
IceCream.new(1).class is *Celery*.
TripleFudge.new(99).class is *Celery*.
TripleFudge.new(99).class is ***TripleFudge***.
Yes!

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 <u>is</u> 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.

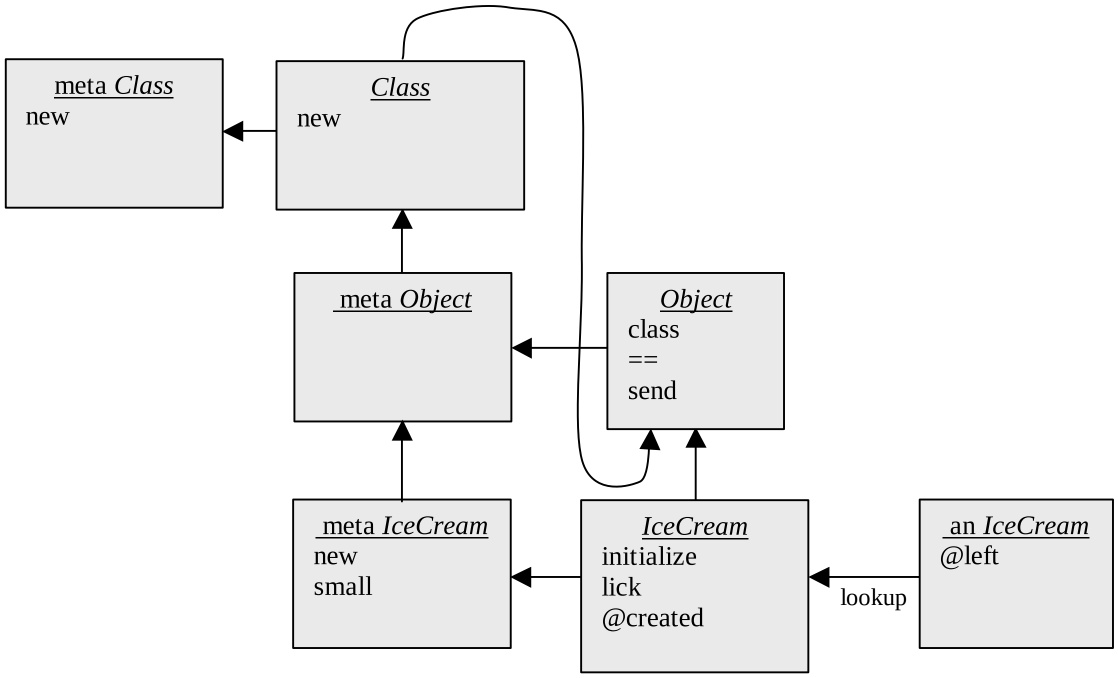
The Tenth Message
In computation, simple rules combine to allow complex possibilities

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*.

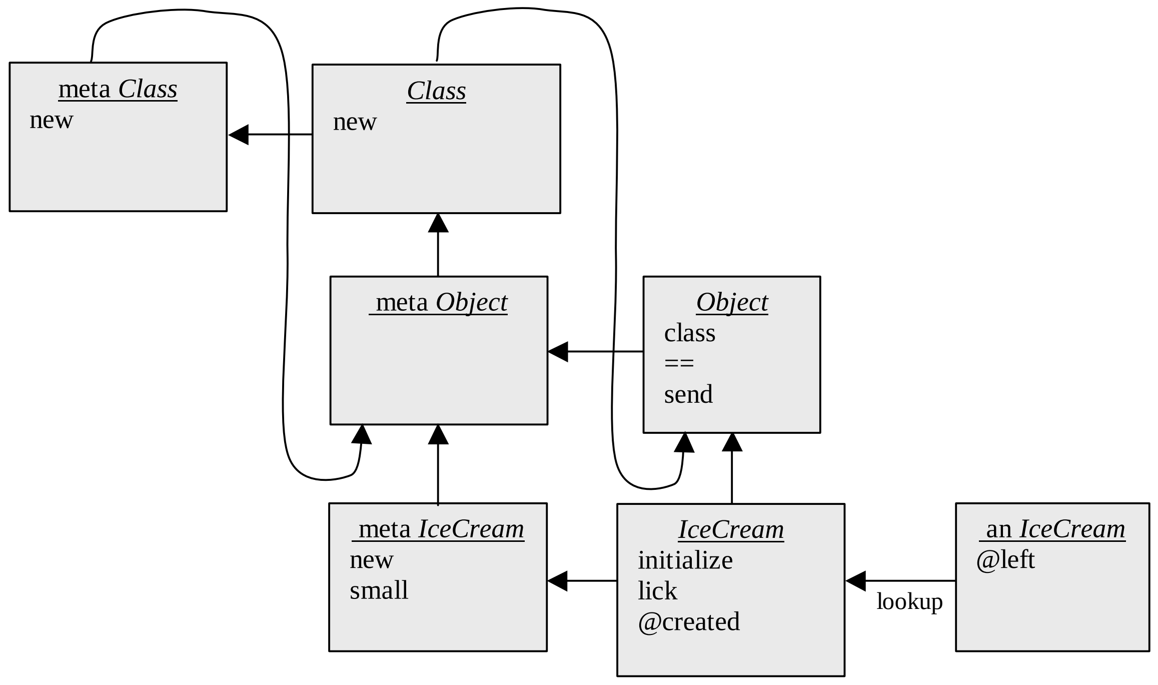
So the arrow up from *Class* should curve back down to *Object*. Don't fix the picture yet.	I want to. I'd rather have clarity than save paper.



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.**