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.

Copyright © 2001 by Brian Marick. All Rights Reserved.

(marick@visibleworkings.com)

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.

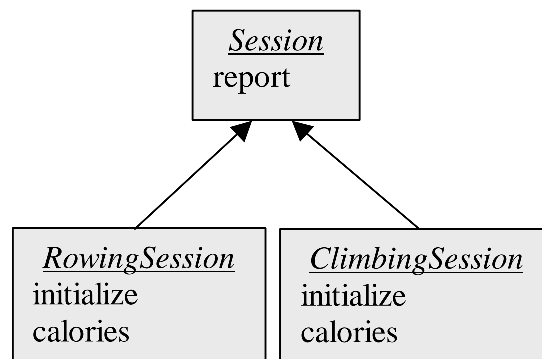
Is method *report* an example of what you want to be common between the two kinds of sessions?

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.

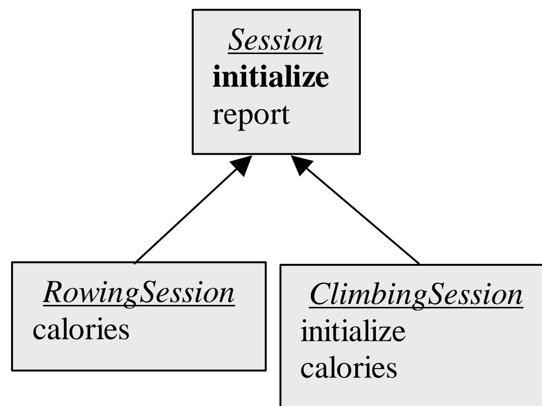
Can you do that for *RowingSession*?

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?

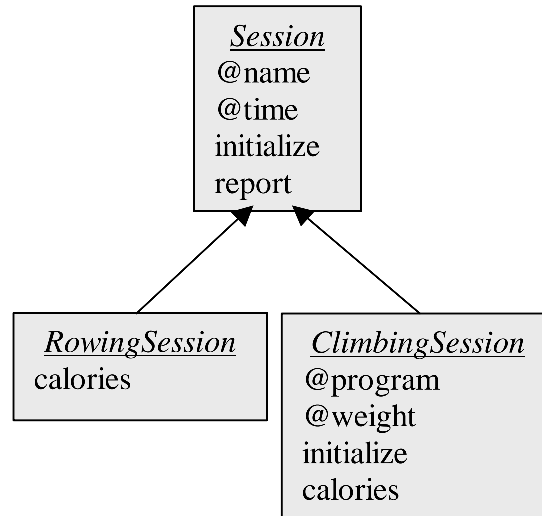


What will happen as a result of this call? *RowingSession.new("buffy", 30)*	The method *new* for the class *RowingSession* will create a *RowingSession* object. Then it will send an *initialize* message to that object. Since *RowingSession* has no *initialize* method, Ruby looks in its superclass, *Session*. 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.

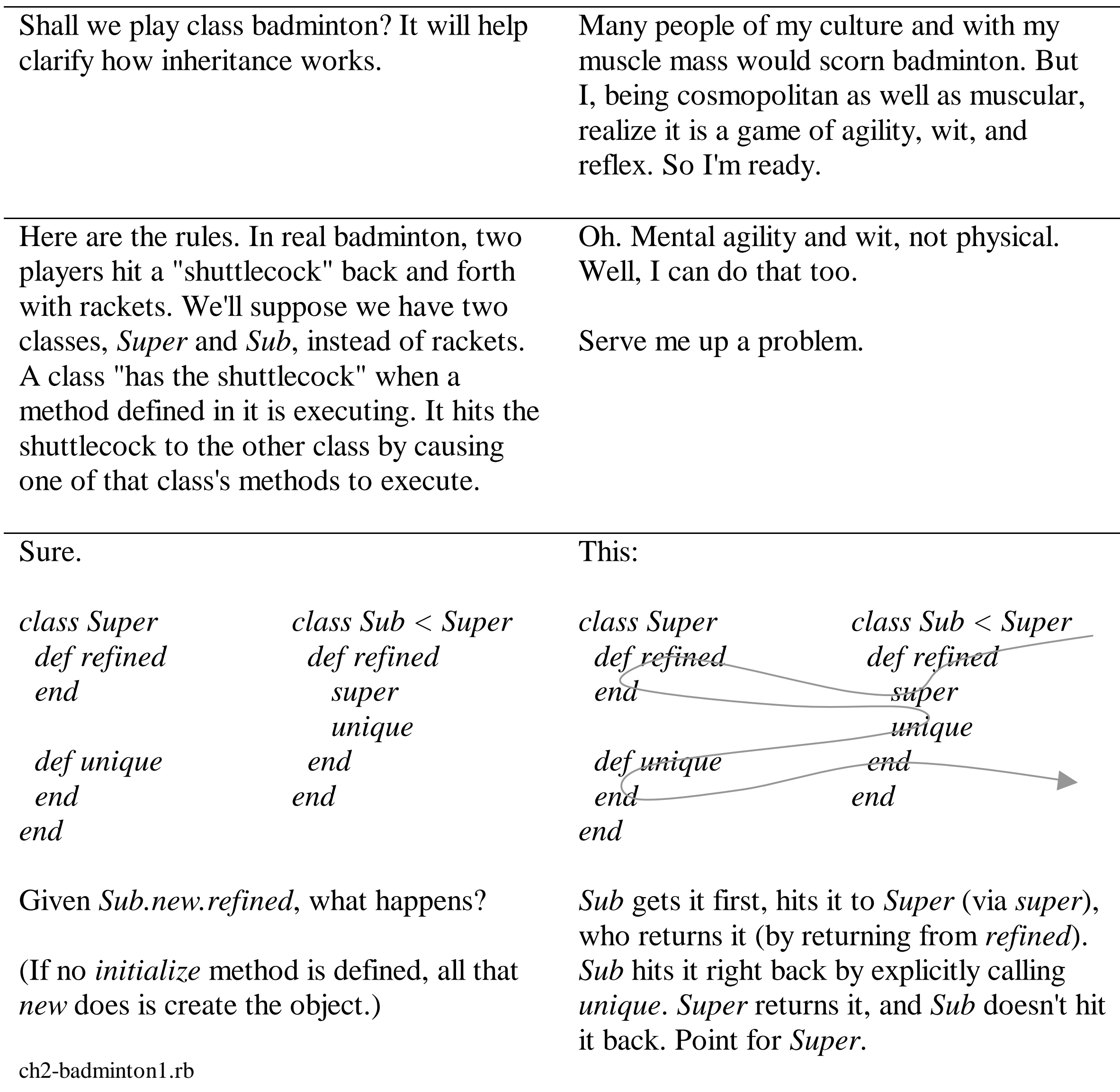
Can you move the duplicate code from *ClimbingSession* to *Session?*	I'm not sure how. Only two of the lines within *ClimbingSession*'s *initialize* method can be moved. The other two lines have to stay, because they set instance variables unique to *ClimbingSessions*:
	
	`class ClimbingSession`
	`  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
  def refined
  end

  def unique
  end
end
```

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

  def unique
  end
end
```

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

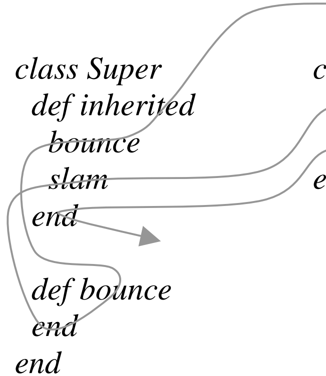
Sub gets it first, hits it to *Super* (via *super*), who returns it (by returning from *refined*). *Sub* hits it right back by explicitly calling *unique*. *Super* returns it, and *Sub* doesn't hit it back. Point for *Super*.

How about this one?

```
class Super
  def inherited
    bounce
    slam
  end

  def bounce
  end
end
```

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




```
class Super
  def inherited
    bounce
    slam
  end

  def bounce
  end
end
```

```
class Sub < Super
   def slam
   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

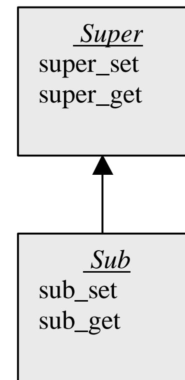
Both *super_get* and *sub_get* answer *"dawn"*.

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:



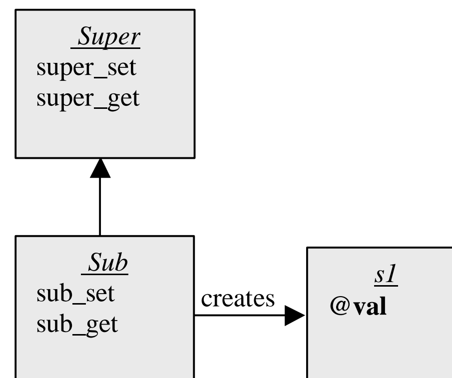
I'm not sure where to put *@val*. It should only go in one place because either class can change it.

Suppose you execute this code:

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

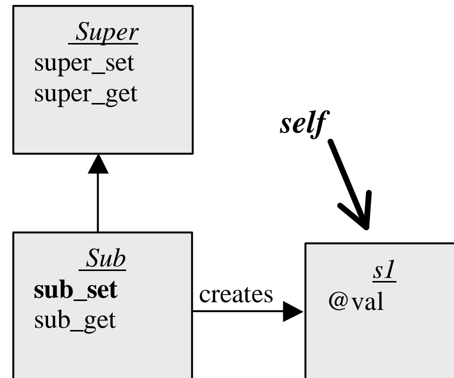
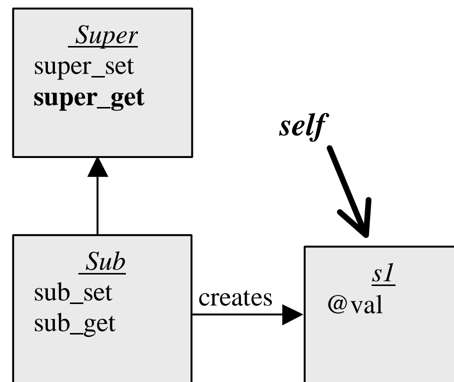
Do the two objects have the same value of *@val*?

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.**