Class definition

Counter: A tally counter

An interesting thing about instance variables

Addition of methods

An interesting thing about class defintions

Sidebar: Fun with eval

Class variables and methods

Setters and getters

A little bit on access control

Getters and setters

A tally counter

Imagine a class named Counter that models a tally counter.

Here's how we might create and interact with an instance of Counter:

```
c1 = Counter.new
c1.click
c1.click
puts c1          # Output: Counter's count is 2
c1.reset

c2 = Counter.new "c2"
c2.click
puts c2          # Output: c2's count is 1

c2.click
printf("c2 = %d\n", c2.count)    # Output: c2 = 2
```

Counter, continued

Here is a partial implementation of Counter:

```
class Counter
   def initialize(label = "Counter")
      @count = 0
      @label = label
   end
end
```

The reserved word class begins a class definition; a corresponding end terminates it. A class name must begin with a capital letter.

The method name initialize is special. It identifies the method that is called when the method new is invoked:

```
c1 = Counter.new

c2 = Counter.new "c2"
```

If no argument is supplied to new, the default value of "Counter" is used.

Obviously, initialize is the counterpart to a constructor in Java.

Counter, continued

For reference:

```
class Counter
   def initialize(label = "Counter")
      @count = 0
      @label = label
   end
end
```

The constructor initializes two instance variables: @count and @label.

Instance variables are identified by prefixing them with @.

An instance variable comes into existence when a value is assigned to it.

Just like Java, each object has its own copy of instance variables.

Unlike variables local to a method, instance variables have a default value of nil.

Counter, continued

For reference:

```
class Counter
   def initialize(label = "Counter")
      @count = 0
      @label = label
   end
end
```

When irb displays an object, the instance variables are shown:

```
>> a = Counter.new "a"
=> #<Counter:0x2c61eb4 @label="a", @count=0>

>> b = Counter.new
=> #<Counter:0x2c4da04 @label="Counter", @count=0>

>> [a,b]
=> [#<Counter:0x2c61eb4 @label="a", @count=0>,
    #<Counter:0x2c4da04 @label="Counter", @count=0>]
```

Counter, continued

Here's the full source:

```
class Counter
   def initialize(label = "Counter")
      @count = 0; @label = label
   end
   def click
      @count += 1
   end
   def reset
      @count = 0
   end
   def count            # Note the convention: count, not get_count
      @count
   end
   def to_s
      return "#{@label}'s count is #{@count}"
   end
end
```

A very common error is to omit the @ on a reference to an instance variable.

An interesting thing about instance variables

Consider this class:

```
class X
   def initialize(n)
      case n
      when 1 then @x = 1
      when 2 then @y = 1
      when 3 then @x = @y = 1
      end
   end
end
```

What's interesting about the following?

```
>> X.new 1          => #<X:0x2c26a44 @x=1>

>> X.new 2          => #<X:0x2c257d4 @y=1>

>> X.new 3          => #<X:0x2c24578 @x=1, @y=1>
```

Addition of methods

In Ruby, a method can be added to an existing class. In the example below we add a label method to Counter, to fetch the value of the instance variable @label.

```
>> c = Counter.new "ctr 1"
=> #<Counter:0x2c26bac @label="ctr 1", @count=0>

>> c.label
NoMethodError: undefined method `label' for #<Counter:0x2c26bac @label="ctr 1",
@count=0>
        from (irb):4
>> class Counter
>>     def label
>>       @label
>>     end
>> end
=> nil

>> c.label
=> "ctr 1"
```

What are the implications of this capability?

Addition of methods, continued

We can extend built-in classes, too!

```
% cat hexstr.rb
class Fixnum
   def hexstr
      return "%x" % self
   end
end
```

Usage:

```
>> load "hexstr.rb"       => true

>> 15.hexstr              => "f"

>> p (10..20).collect { |n| n.hexstr }
["a", "b", "c", "d", "e", "f", "10", "11", "12", "13", "14"]
=> nil
```

An interesting thing about class definitions

Observe the following. What does it suggest to you?

```
>> class X
>> end
=> nil

>> p (class X; end)
nil
=> nil

>> class X; puts "here"; end
here
=> nil
```

Class definitions are executable code

In fact, a class definition is executable code. Consider the following, which uses a case statement to selectively execute defs.

```
class X
   print "What methods would you like? "
   methods = gets.chomp
   methods.each_byte { |c|
      case c
      when ?f then def f; "from f" end
      when ?g then def g; "from g" end
      when ?h then def h; "from h" end
      end
      }
end
```

Execution:

```
What methods would you like? fg
>> c = X.new                              => #<X:0x2c2a1e4>
>> c.f                                    => "from f"
>> c.h
NoMethodError: undefined method `h' for #<X:0x2c2a1e4>
```

Sidebar: Fun with eval

Kernel#eval **parses a string containing Ruby source code and executes it.**

```
>> s = "abc"                    => "abc"

>> n = 3                        => 3

>> eval "x = s * n"             => "abcabcabc"

>> x                            => "abcabcabc"

>> eval "x[2..-2].length"       => 6

>> eval gets
s.reverse
                                => "cba"
```

Look carefully at the above. Note that eval uses variables from the current environment and that an assignment to x is reflected in the environment.

Bottom line: A Ruby program can generate code for itself.

Sidebar, continued

Problem: Create a file new_method.rb with a class X that prompts the user for a method name, parameters, and method body. It then creates that method. Repeat.

```
>> load "new_method.rb"
What method would you like? add
Parameters? a, b
What shall it do? a + b
Method add(a, b) added to class X

What method would you like? last
Parameters? a
What shall it do? a[-1]
Method last(a) added to class X

What method would you like? ^D

>> c = X.new            => #<X:0x2c2980c>

>> c.add(3,4)           => 7

>> c.last [1,2,3]       => 3
```

Sidebar, continued

Solution:

```
class X
   while true
      print "What method would you like? "
      name = gets || break
      name.chomp!

      print "Parameters? "
      params = gets.chomp

      print "What shall it do? "
      body = gets.chomp

      code = "def #{name} #{params}; #{body}; end"

      eval(code)
      print("Method #{name}(#{params}) added to class #{self}\n\n");
   end
end
```

Is this a useful capability or simply fun to play with?

Class variables and methods

Just as Java, Ruby provides a way to associate data and methods with a class itself rather than each instance of a class.

Java uses the static keyword to denote a class variable.

In Ruby a variable prefixed with two at-signs is a class variable.

Here is Counter augmented with a class variable that keeps track of how many counters have been created:

```
class Counter
   @@created = 0              # Must precede any use of @@created

   def initialize(label = "Counter")
      @count = 0; @label = label
      @@created += 1
   end

end
```

Note: Unaffected methods are not shown.

Class variables and methods, continued

To define a class method, simply prefix the method name with the name of the class:

```
class Counter
   @@created = 0

   ... other methods ...

   def Counter.created          # class method
      return @@created
   end
end
```

Usage:

```
>> Counter.created             => 0
>> c = Counter.new             => #<Counter:0x... @label="Counter", @count=0>
>> Counter.created             => 1
>> 5.times { Counter.new }     => 5
>> Counter.created             => 6
```

A little bit on access control

By default, methods are public. If private appears on a line by itself, subsequent methods in the class are private.

```
class X
   def f; puts "in f"; g end        # Note: calls g

  private
   def g; puts "in g" end
end

>> x = X.new                 => #<X:0x2c0cc84>
>> x.f
in f
in g

>> x.g
NoMethodError: private method `g' called for #<X:0x2c0cc84>
```

In Ruby, there is simply no such thing as a public class variable or public instance variable. All access must be through methods.

Getters and setters

If Counter were in Java, we might provide methods like void setCount(int n) and int getCount().

In Counter we provide a method called count to fetch the count.

Instead of something like setCount, we'd do this:

```
def count= n          # IMPORTANT: Note the trailing '='
    print("count=(#{n}) called\n")
    @count = n unless n < 0
end
```

Usage:

```
>> c = Counter.new      => #<Counter:0x2c94094 @label="Counter", @count=0>

>> c.count = 10
count=(10) called

>> c                    => #<Counter:0x2c94094 @label="Counter", @count=10>
```

Getters and setters, continued

Here's class to represent points on a 2d Cartesian plane:

```
class Point
   def initialize(x, y)
      @x = x
      @y = y
   end
   def x; @x end
   def y; @y end
end
```

Usage:

```
>> p1 = Point.new(3,4)        => #<Point:0x2c72c78 @x=3, @y=4>

>> [p1.x, p1.y]               => [3, 4]
```

It can be tedious and error prone to write a number of simple getter methods, like Point#x and Point#y.

Getters and setters, continued

The method attr_reader *creates* getter methods. Here's an equivalent definition of Point:

```
class Point
   def initialize(x, y)
      @x = x
      @y = y
   end
   attr_reader :x, :y        # Recall that :x and :y are Symbols.  (But "x" and "y" work!)
end
```

Usage:

```
>> p = Point.new(3,4)    => #<Point:0x2c25478 @x=3, @y=4>

>> p.x                   => 3

>> p.y                   => 4

>> p.x = 10
NoMethodError: undefined method `x=' for #<Point:0x2c29924 @y=4, @x=3>
```

Why does p.x = 10 fail?

Getters and setters, continued

If you want both getters and setters, use attr_accessor:

```
class Point
   def initialize(x, y)
      @x = x
      @y = y
   end

   attr_accessor :x, :y
end
```

Usage:

```
>> p = Point.new(3,4)        => #<Point:0x2c298d4 @y=4, @x=3>
>> p.x                       => 3
>> p.y = -20                 => -20
>> p                         => #<Point:0x2c298d4 @y=-20, @x=3>
```

It's important to appreciate that attr_reader and attr_accessor are *methods that create methods*. We could define a method called getters that has the same effect as attr_reader.