COMP 9 / EN47 - Exploring Computer Science

Lecture 8: Object-Oriented Programming

March 8, 2011

1 Introduction

Object-oriented programming is a way to *organize* your code, *encapsulate* information so you don't have to pay attention to all of it, and make writing *reusable* code more convenient

2 Classes seen so far

Thus far, we have used several built-in Ruby types. They're really what we call *classes*.

- `Float`
- `Integer`
- `String`
- `Array`
- `Hash`
- `NilClass`
- `File`

We've also used a couple of classes that are part of BioRuby

- `Bio::Blast`
- `Bio::Blast::Report`
- `Bio::Blast::Report::Hit`

- `Bio::FastaFormat`

We've seen some *methods* defined on these classes

- `2+2`
- `"Hello, " + "World"`
- `[] << 3`
- `chimp_factory.query(gene)`
- `File.open(outfile, 'w')`

To create a new *instance* of a class, we'll typically use the `new` method:

- `Time.new`
- `Array.new` is the same as `[]`
- `String.new` is the same as `''`
- `File.open` is an exception!
  - It creates a new file *object, not necessarily a new file on disk!
- `Bio::Blast.local` is also an exception
  - It's a specialized version of `Bio::Blast.new`
  - and as opposed to `Bio::Blast.remote`

3 Creating our own classes

We have represented playing cards as hashes or arrays

```
{:suit => 'Club', :name => 'King', :value => 10}
```

```
['Club', 'King', 10]
```

These are perfectly useful for gathering together pieces of *information*, such as the suit or value of a card. However, things get confused when we add in methods. What if we want a method to give us the value of a *hand*, and also a method to give us the value of a *card*? We could call them `value_of_hand` and `value_of_card` but pretty soon a program might be littered with methods.

3.1 Classes to the rescue!

Classes let us encapsulate *methods* as well. Consider:

```
class Card
    attr_accessor :suit, :name
    attr_reader :value

    def face?
        ['Jack', 'Queen', 'King'].include?(@name)
    end

    def get_value
        values = ['Ace', 'Two', 'Three', 'Four', 'Five', 'Six',
            'Seven', 'Eight', 'Nine', 'Ten']
        v = values.index(@name)
        if v
            return v + 1
        elsif face?
            return 10
        else
            raise "Invalid value!"
        end
    end

    def initialize(suit, value)
        @suit, @name = suit, name
        @value = get_value
    end
end

c = Card.new('Clubs', 'Three')
puts c.value     # ==> 3
```

3.1.1 About that weird @ sign

The `@` sign is an *instance variable*, and it *persists* through the lifetime of an instance of a class. Different methods within the class can access it, so its *scope* is greater than those of the variables (local variables) you've seen so far.

3.2 Ruby classes are *open*

This means we can add methods to already defined classes!

For blackjack, perhaps we want to add a `handle_ace` method

```
class Card
    def handle_ace
        if @name == 'Ace'
            puts "You have an ace. 1 or 11?"
            answer = gets.to_i
            if answer == 1 or answer == 11
                @value = answer
            else
                puts "cheater!"
            end
        end
    end
end
```

Now, what about adding an `average` method to the `Array` class?

```
class Array
    def average
        numerator = inject(0.0){|s,e| s += e}
        denominator = length
        numerator/denominator
    end
end

[1,3,4].average # ==> 2.6666666666666666667
```

When does this break horribly?

```
['a', 'hi', 'bye'].average # ==> TypeError: String can't be coerced into Float
```

But sometimes it's convenient, or really useful.

3.3 Deck example

```
class Deck

    def initialize
        @cards = []

        values = ['Ace', 'Two', 'Three', 'Four', 'Five', 'Six',
            'Seven', 'Eight', 'Nine', 'Ten' 'Jack', 'Queen', 'King']
        suits = ['Spades', 'Diamonds', 'Hearts', 'Clubs']
        deck = []
```

```
        suits.each do |s|
            values.each do |v|
                deck << Card.new(s, v)
            end
        end
    end

    def shuffle!
        @cards = @cards.sort_by{rand}
    end

    def deal
        c = @cards.delete_at(0)
        c.handle_ace
        return c
    end

    def value
        @cards.inject(0){|card| card.value}
    end

end

def Hand

    def initialize
        @cards = []
    end

    def value
        @cards.inject(0){|card| card.value}
    end

    def busted?
        value >= 21
    end

    def winning?
        value == 21
    end

    def stick?
        puts "Hit or stick?"
        answer = gets.chomp
        if answer == 'Stick'
            return true
```

```
        else
            return false
        end
    end

end
```

Ok, this is complicated, but now look how simple blackjack can be!

```
deck = Deck.new
hand = Hand.new
deck.shuffle!
2.times do
    hand << deck.deal
end

if not hand.busted? and not hand.stick?
    hand << deck.deal
end

if hand.winning?
    puts "Winner!"
else
    puts "Sorry... your score is #{hand.value}"
end
```

4 Class vs. Instance methods

There are two types of methods you will likely encounter.

Class methods are ones that *always* start with the name of the class, like `File.open`, `Deck.new`, or `Bio::Blast.local`. Often they create a new *instance* of the class: a specific string or number or filehandle. Or they might be utilities like `File.exist?` that don't actually need to create a specific instance.

Instance methods are ones that deal with a *specific instance* of a class