Week 1 Section - Pair Programming in Ruby

Part One: What Would Ruby Do?

Find a partner and begin typing the following exercises into the interpreter. You should alternate who types and who explains the output.

1)
```
fruit1 = “strawberry”
fruit2 = “banana”
puts fruit1.reverse
=> “yrrebwarts”
puts fruit2.reverse!
=> “ananab”
fruit1 + “ ” + fruit2
=> “strawberry ananab”
```

2)
```
class String
  @@hello = “hi there!”
  def hello; “world”; end
end
“smoothie”.hello
=> “world”
```

3)
```
class Fruit
  def method_missing(meth)
    if meth.to_s =~ /^tastes_(.+)\?$/
      "Yup, that fruit tastes #{$1}!"
    else
      super
    end
  end
end

orange = Fruit.new
orange.bitter?
NoMethodError
orange.tastes_sour?
=> “Yup, that fruit tastes sour!”
orange.tastes_sweet?
=> “Yup, that fruit tastes sweet!”
```

Note that by convention, exclamation marks in ruby method names often indicate that the method will mutate the object it’s being called on. This is why fruit2 in the last line of example one returns ananab again—we called reverse! on fruit2, whereas we only called reverse on fruit1. fruit2 was mutated; fruit1 was not.

Prefixing a variable with @@ defines it as a class variable. Prefixing it with only @ defines it as an instance variable. One must create methods that interact with these variables (e.g. getter and setter methods) in order to access them. Dot notation in ruby exclusively makes method calls; there has only been one “hello” method defined in example two, and thus this is what is called.

Part Two: Collections

In this next part, try to rewrite each of the following method as one (short) line. One person should be the **writer**, while the other person **explains what to write**. Try alternating roles between the two exercises. (Hint: see figure 3.7 in the textbook.)

1)
```
def foo(arr)
  res = 0
  arr.each do |n|
    res += n
  end
  res
end
```

2)
```
def bar(hsh)
  res = {}
  hsh.each do |k, v|
    if v > 100
      res[k] = v
    end
  end
  res
end
```

1) def foo(arr); arr.reduce(:+);end
2) def bar(hsh); hsh.select { |k, v| v > 100}; end

Part Three: Iterators

In this part, create your own iterators with the yield statement that return the following elements. Again, alternate roles between the two exercises.

Write a function `fib(n)` that yields the first n Fibonacci numbers in sequence and returns nil.

```
>> fib(4) { |x| puts x }
1
1
2
3
nil
```

```
def fib(n)
  prev, curr = 0, 1
  n.times do
    yield curr
    prev, curr = curr, prev + curr
  end
end
```

Write the function `Array#odds` which yields the odd-indexed elements of the array in sequence and returns nil.

```
>> [10, 30, 50, 70, 90].odds do |n|
..   puts n
.. end
30
70
nil
```

```
class Array
  def odds
    self.each_with_index do |val, index|
      if index % 2 == 1
        yield val
      else
        next
      end
    end
    nil
  end
end
```

Extra Practice

Implement a linked list. Try to include the add, delete, and contains operations.
The following is one possible implementation (albeit sub-optimal).

```
class ListNode
  attr_accessor :next
  attr_reader :value
  def initialize value
    @value = value
    @next = nil
  end
end
```

```
class LinkedList
  def initialize
    @head = nil
  end

  def add value
    if @head.nil?
      @head = ListNode.new value
    else
      node = @head
      node = node.next while node.next
      node.next = ListNode.new value
    end
  end

  def contains value
    node = @head
    while node
      if node.value == value
        return true
      end
      node = node.next
    end
    return false
  end

  def delete value
    if @head.value == value
      @head = @head.next
      return true
    end
    node = @head
    while node = node.next
      if node.next and node.next.value == value
        node.next = node.next.next
        return true
      end
    end
    return false
  end
end
```