**Double Dispatch / Inheritance:** Rock, Paper, Scissors Example

Write three classes, `Rock`, `Paper` and `Scissors`. They can all add a RPSObject mixin or be subclasses of some superclass, but that isn't necessary for this example.

OOP Approach**:** The client code should be able to call "`a.fights(b)`" for some arbitrary R/P/S objects a and b (without necessarily knowing whether the objects a and b are rocks, papers or scissors). Implement this functionality using double dispatch.
(Hint: you should be adding 12 methods, 4 per class.)

**(this topic (double dispatch) won't be on the final. To avoid possible confusion, send us an email if you are curious)**

Functional Approach: Now implement this using a functional programming approach.

Solution:

- OOP: Add a "`fights(other)`" method, as well as "`fightsRock(rock)`", "`fightsPaper(paper)`", and "`fightsScissors(scissors)`" methods to each of the three classes (total of 12 methods). The `fights(other)` method in `class X` should call `other.fightsX(self)`. Within each `fightsX(obj)` method, the correct string can be returned.
- Functional: Some if/else logic to check the types of the two objects (if a is_a X and b is_a Y then "win" etc). Each fights method can call the static method with self and obj, or just do the type checking on the other object within its fights method.

```
class RPSObject
end

class Rock < RPSObject

  def fight other
    other.fightRock
  end

  def fightRock
    "tie"
  end

  def fightPaper
    "win"
  end

  def fightScissors
    "lose"
  end

  def to_s
    "Rock"
  end

end

class Paper < RPSObject

  def fight other
    other.fightPaper
  end

  def fightRock
    "lose"
  end

  def fightPaper
    "tie"
  end

  def fightScissors
    "win"
  end

  def to_s
    "Paper"
  end

end

class Scissors < RPSObject
 def fight other
    other.fightScissors
  end

  def fightRock
    "win"
  end

  def fightPaper
    "lose"
  end

  def fightScissors
    "tie"
  end

  def to_s
    "Scissors"
  end
end

# Testing
a = [Rock.new, Paper.new, Scissors.new]
a.combination(2).to_a.each { |a,b| puts (a.fight b) }
```

## Class and Mixins and Coerce:

1: implement Comparable and override compareTo method
2: include Comparable and define the method **<=>**
3:  (this is really a bad example, read Ruby's Enumerable class for more information)

```
def <=> other
      return @nume * other.deno <=> other.nume * @deno
end
```

1: It means we can take element one by one from the object, like using a for-each loop
2: In Java, it's more close to iterable. By implement iterable
3: include Enumerable and define the method **each**
4:

```
def each
      yield @nume
      yield @deno
end
```

1: coerce means using dispatch to convert an object to the one that supports such operation
2:

```
def coerce n
     return [PosRational.new(n), this]
end
```

3: (not important) coerce cannot apply to things other than operators
Extra: Java does not allow operator overload

## Extra practice questions:

1:

```
def min
     minimum = nil
     each {|x| minimum = x if minimum.nil? or x < minimum}
     min
end
```

2:

```
def min2
     first = nil
     second = nil
     each {|x| first = x if first.nil? or x < first}
     each {|x| second = x if second.nil? and x > first or x > first
and x < second}
     second
end
```