In this lecture, we will discuss…

- ✧ Modules
  - As Namespaces
  - As Mixins
- ✧ Using built-in Ruby modules, especially `Enumerable`
- ✧ `require_relative`

Module

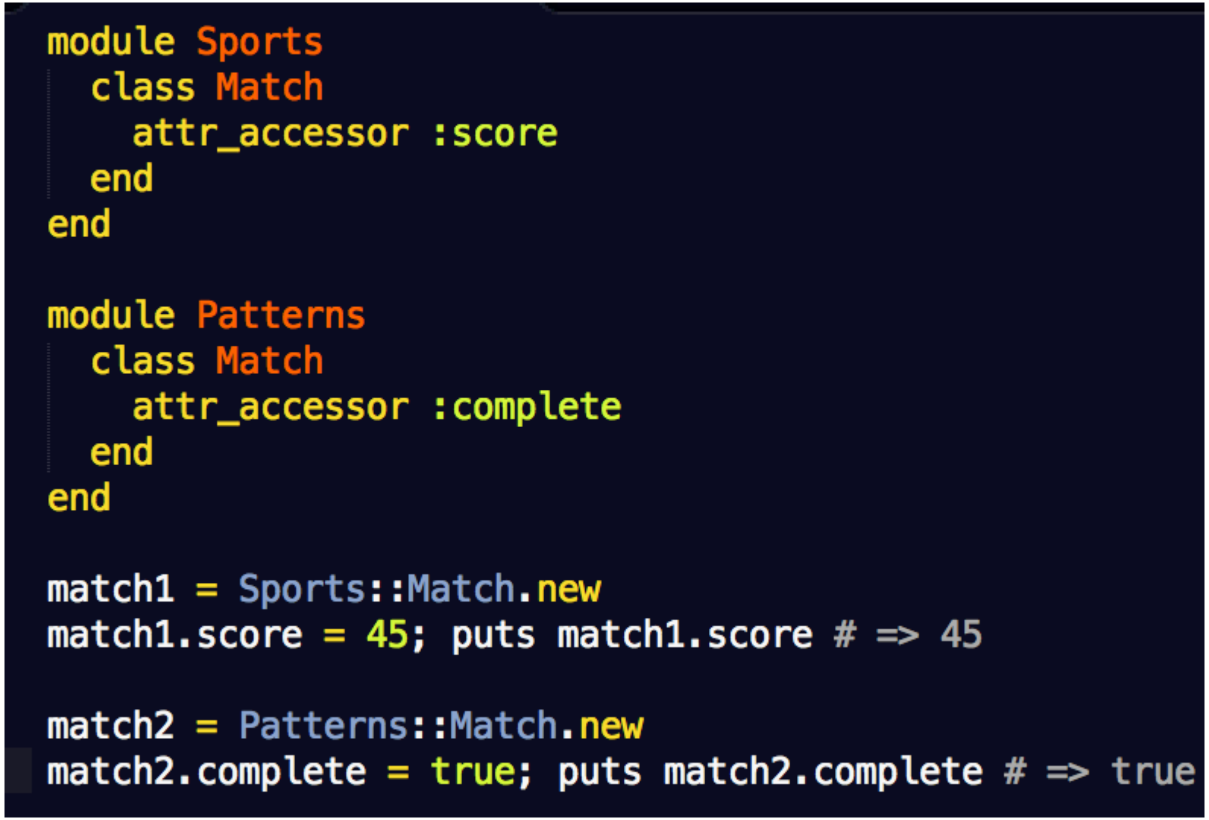
- ✧ **Container** for classes, methods and constants
  - Or other **modules**…
- ✧ Like a `Class`, but cannot be **instantiated**
  - `Class` inherits from `Module` and adds `new`

Module

✧ Serves **two** main purposes:

1. Namespace
2. Mix-in

Module as Namespace



```
module Sports
  class Match
    attr_accessor :score
  end
end

module Patterns
  class Match
    attr_accessor :complete
  end
end

match1 = Sports::Match.new
match1.score = 45; puts match1.score # => 45

match2 = Patterns::Match.new
match2.complete = true; puts match2.complete # => true
```

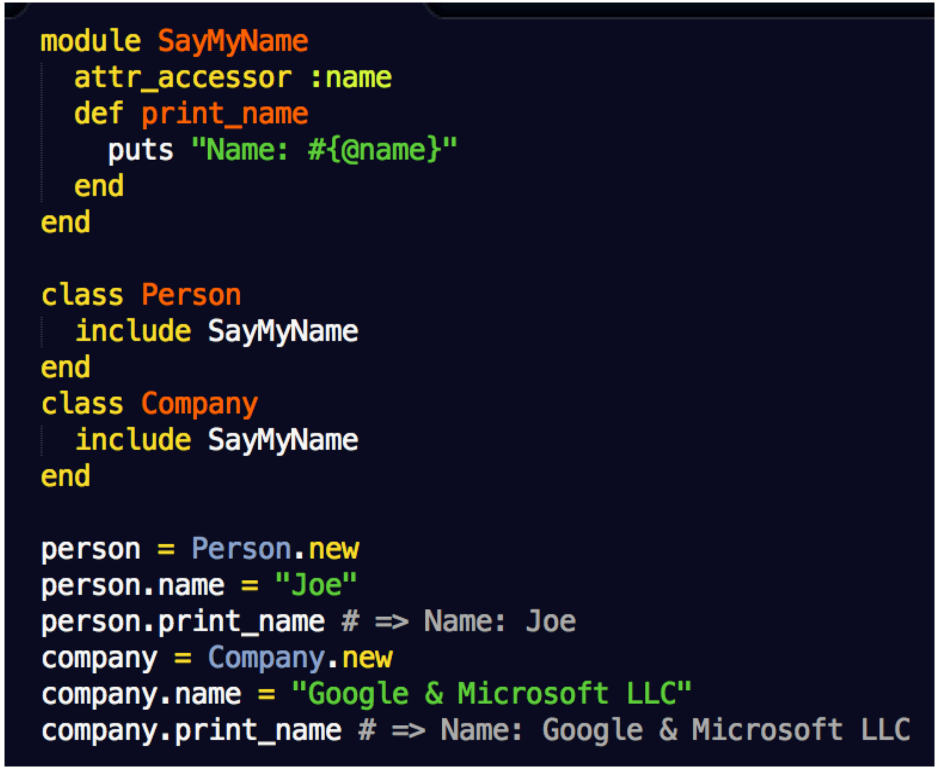
Note the use of :: operator

Module as Mixin

- ✧ Interfaces in **OO**
  - **Contract** - define what a class **“could”** do
- ✧ Mixins provide a way to **share** (mix-in) ready code among **multiple classes**

You can include built-in modules like `Enumerable` that can do the hard work for you!

Module as Mixin



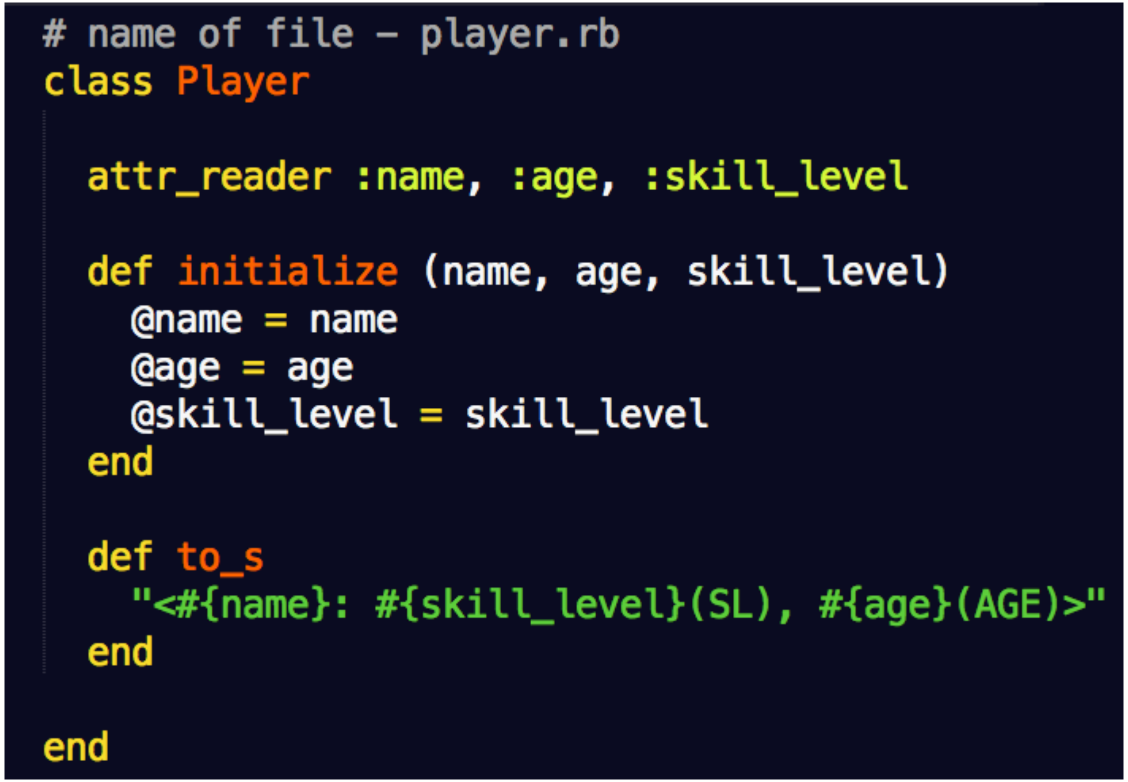
```ruby
module SayMyName
  attr_accessor :name
  def print_name
    puts "Name: #{@name}"
  end
end

class Person
  include SayMyName
end
class Company
  include SayMyName
end

person = Person.new
person.name = "Joe"
person.print_name # => Name: Joe
company = Company.new
company.name = "Google & Microsoft LLC"
company.print_name # => Name: Google & Microsoft LLC
```

Enumerable Module

- `map`, `select`, `reject`, `detect` etc.
- Used by `Array` class and **many others**
- You can **include it** in your **own class**
- Provide an **implementation** for `each` method

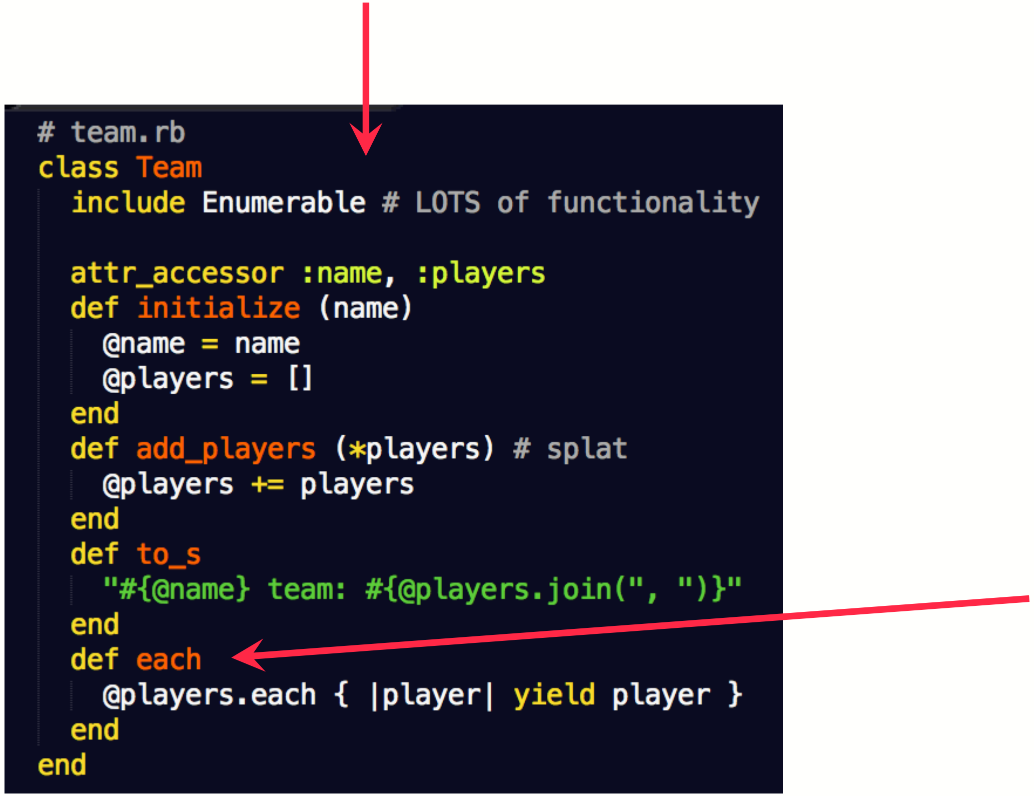
All the other functionality of `Enumerable` is magically available to you!

Player



```ruby
# name of file - player.rb
class Player

  attr_reader :name, :age, :skill_level

  def initialize (name, age, skill_level)
    @name = name
    @age = age
    @skill_level = skill_level
  end

  def to_s
    "<#{name}: #{skill_level}(SL), #{age}(AGE)>"
  end

end
```

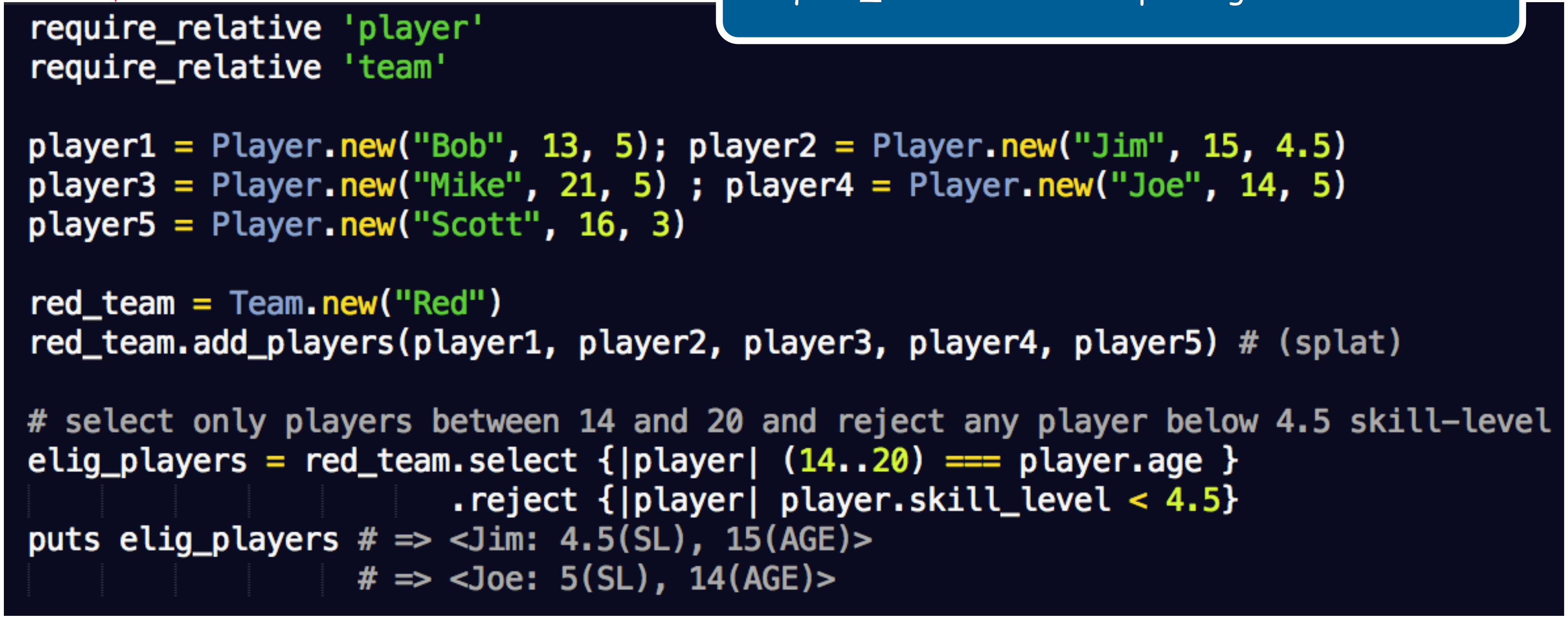
Team



```ruby
# team.rb
class Team
  include Enumerable # LOTS of functionality

  attr_accessor :name, :players
  def initialize (name)
    @name = name
    @players = []
  end
  def add_players (*players) # splat
    @players += players
  end
  def to_s
    "#{@name} team: #{@players.join(", ")}"
  end
  def each
    @players.each { |player| yield player }
  end
end
```

Enumerable in Action

require_relative allows importing other .rb files!!!



```ruby
require_relative 'player'
require_relative 'team'

player1 = Player.new("Bob", 13, 5); player2 = Player.new("Jim", 15, 4.5)
player3 = Player.new("Mike", 21, 5) ; player4 = Player.new("Joe", 14, 5)
player5 = Player.new("Scott", 16, 3)

red_team = Team.new("Red")
red_team.add_players(player1, player2, player3, player4, player5) # (splat)

# select only players between 14 and 20 and reject any player below 4.5 skill-level
elig_players = red_team.select {|player| (14..20) === player.age }
                       .reject {|player| player.skill_level < 4.5}
puts elig_players # => <Jim: 4.5(SL), 15(AGE)>
                  # => <Joe: 5(SL), 14(AGE)>
```

Summary

- ✧ Modules allow you to **“mixin” useful code** into other **classes**
- ✧ `require_relative` is useful for **including other ruby files** relative to the **current ruby code**

What’s next?

- ✧ Scope