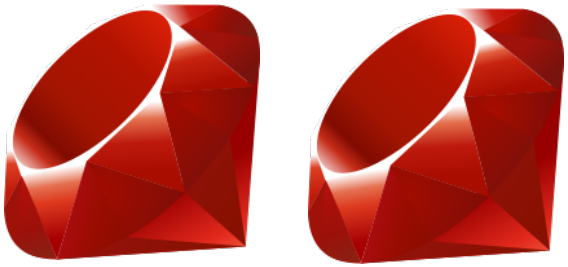
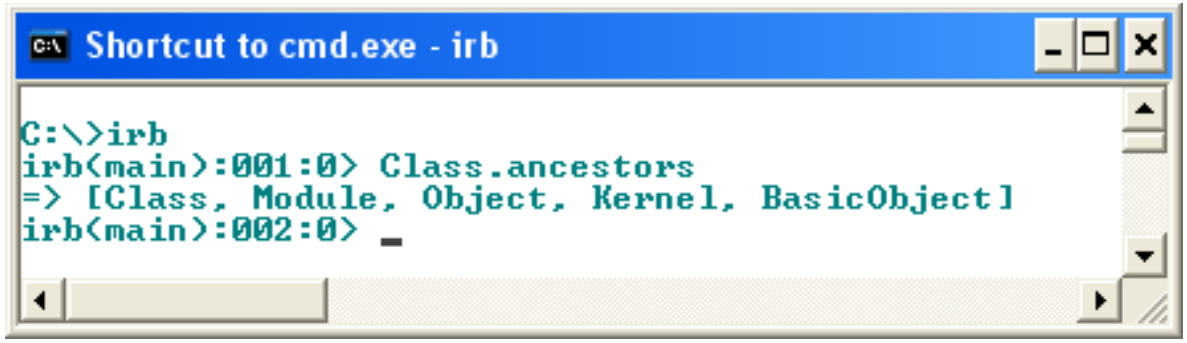


Modules

Module

- Container for classes, methods and constants (or other modules)
- Like a `Class`, but cannot be instantiated
  - `Class` inherits from `Module` and adds `new`
- Serves 2 main purposes:
  1. Namespace
  2. Mix-in



Modules as namespaces

```
module Toolbox
  class Ruler
    attr_accessor :length
  end
end

module Country
  class Ruler
    attr_accessor :name
  end
end

ruler1 = Toolbox::Ruler.new
ruler1.length = 20
puts ruler1.length # => 20

ruler2 = Country::Ruler.new
ruler2.name = "Czar Nicolai"
puts ruler2.name # => Czar Nicolai
```

Note the use of :: operator

Modules as Mixins

- Think of Java interfaces
  - Contract - define what a class “could” do
- Mixins are WAAAY cooler
  - Provide a way to share (mix-in) ready code among multiple classes
- You can include built-in modules like `Enumerable` and `Comparable` which can do the hard work for you!

Modules as Mixins - example

```
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!
- Only need to provide an implementation for `each` method and all the other functionality of `Enumerable` is magically available to you!

Enumerable - example

```
# 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.each { |player| @players << player }
  end
  def to_s
    "#{@name} team: #{@players.join(", ")}"
  end
  def each
    @players.each { |player| yield player } # yield explained later
  end
end
```

yield player to a block (explained later)

Enumerable example (continued)

```
require_relative "team" #load in team.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

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}
p elig_players # => [<Jim: 4.5(SL), 15(AGE)>, <Joe: 5(SL), 14(AGE)>]
```

Comparable example

- Provides a class with behavior for comparison operators
- `<, >, <=, >=, ==, between?`
- Only need to implement `<=>` to use it in your own class

Comparable example

```
class Person
  include Comparable
  attr_accessor :name, :age
  def initialize (name, age)
    @name = name
    @age = age
  end
  def <=> (other)
    @age <=> other.age
  end
end

people = [Person.new("Joe", 13), Person.new("Joel", 10), Person.new("Rich", 11)]
puts "Unsorted people: #{people.map(&:name)}"
# => Unsorted people: ["Joe", "Joel", "Rich"]
puts "Sorted people: #{people.sort.reverse.map(&:name)}"
# => Sorted people: ["Joe", "Rich", "Joel"] (descending)
puts people[2].between?(people[1], people[0]) # => true
```

include vs. extend

- `include` includes module's methods as instance methods
- What if you want to include module's methods as class methods?
- Use `extend` instead
- To add some of module's methods as instance and others as class methods – see PickAxe (page ~384)

extend example

```
module Finder
  def find_by_name(name) # case-insensitive find by name
    things.detect {|thing| thing.name.downcase == name.downcase}
  end
end
class Dog
  extend Finder # Makes Finder's methods class methods for this class
  attr_accessor :name
  def initialize(name)
    Dog.things << self # Add any instance created to class method things
    @name = name
  end
  def self.things
    @@things ||= [] # Keep track of all dog instances created
  end
end
Dog.new("Red"); Dog.new("Blue"); Dog.new("Green")
puts Dog.find_by_name("blue").name # => Blue
```

Calls class method `things` (when *extended* as opposed to *included*)