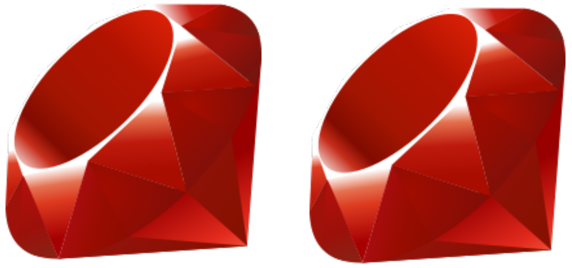

Classes

Ruby OO

- Identify things your program is dealing with
- Classes are things
  - Containers of methods (*behavior*)
- Objects are instances of those things
  - Instance variables (*state*)
    - Begin with @ - ex. `@name`
    - Not declared; spring into existence when first used
    - Available to all instance methods of the class

Classes

- Classes are factories
  - Calling `new` method creates an instance of class
  - Actually, `new` causes `initialize` to be called when creating an instance of a class
- Object's state can be (should be) initialized inside the `initialize` method, the "constructor"

Classes

```
class Person
  def initialize (name, age) # CONSTRUCTOR
    @name = name
    @age = age
  end
  def get_info
    @additional_info = "Interesting"
    "Name: #{@name}, age: #{@age}"
  end
end

person1 = Person.new("Joe", 14)
p person1.instance_variables # [:@name, :@age]
puts person1.get_info # => Name: Joe, age: 14
p person1.instance_variables # [:@name, :@age, :@additional_info]
```

Getting / Setting Data

- Instance variables are private
  - Cannot be accessed from outside the class
- Methods have public access by default
- To access instance variables – need to define “getter” / “setter” methods

Getting / Setting Data (Continued)

```
class Person
  def initialize (name, age) # CONSTRUCTOR
    @name = name
    @age = age
  end
  def name
    @name
  end
  def name= (new_name)
    @name = new_name
  end
end

person1 = Person.new("Joe", 14)
puts person1.name # Joe
person1.name = "Mike"
puts person1.name # Mike
puts person1.age # undefined method `age' for #<Person:
```

Getting / Setting Data (Continued)

- Many times the getter/setter logic is simple
  - Get existing value / Set new value
- There should be an easier way instead of actually defining the getter/setter methods…
- Use `attr_*` form instead
  - `attr_accessor` – getter and setter
  - `attr_reader` – getter only
  - `attr_writer` – setter only

Getting / Setting Data (Continued)

```
class Person
  attr_accessor :name, :age # getters and setters for name and age
end

person1 = Person.new
p person1.name # => nil
person1.name = "Mike"
person1.age = 15
puts person1.name # => Mike
puts person1.age # => 15
person1.age = "fifteen"
puts person1.age # => fifteen
```

Getting / Setting Data (Continued)

- 2 problems with the previous example
  1. Person is in an uninitialized state upon creation
     - Without a name or age
  2. We probably want to control the max age assigned
- **Solution:** Use a constructor and a more intelligent age setter

self

- When inside instance method, `self` (similar to `this` in Java) refers to the object itself
- Usually, using `self` for calling other methods of the same instance is extraneous
- At other times – using `self` is required
  - When it could mean a local var assignment
- Outside instance method definition – `self` refers to the class itself! (*…Discussed later…*)

Getting / Setting Data (Continued)

```
class Person
  attr_reader :age
  attr_accessor :name

  def initialize (name, ageVar) # CONSTRUCTOR
    @name = name
    self.age = ageVar # call the age= method
    puts age
  end
  def age= (new_age)
    @age = new_age unless new_age > 120
  end
end

person1 = Person.new("Kim", 13) # => 13
puts "My age is #{person1.age}" # => My age is 13
person1.age = 130 # Try to change the age
puts person1.age # => 13 (The setter didn't allow the change)
```

Why do we need `self` here?

var = var || something

- || operator evaluates the left side
  - If true – returns it
  - Else – returns the right side
  - `@x = @x || 5` will return `5` the first time and `@x` the next time
- Short form
  - `@x ||= 5` – same as above

||= example

```
class Person
  attr_reader :age
  attr_accessor :name

  def initialize (name, age) # CONSTRUCTOR
    @name = name
    self.age = age # call the age= method
  end
  def age= (new_age)
    @age ||= 5 # default
    @age = new_age unless new_age > 120
  end
end
person1 = Person.new("Kim", 130)
puts person1.age # => 5 (default)
person1.age = 10 # change to 10
puts person1.age # => 10
person1.age = 200 # Try to change to 200
puts person1.age # => 10 (still)
```

Only set to 5 the first time

Class methods and variables

- Invoked on the class (as opposed to instance of class), similar to `static` methods in Java
- `self` OUTSIDE of the method definition refers to the `Class` object
- 3 ways to define class methods
- Class variables begin with `@@`

Class methods and vars example

```
class MathFunctions
  def self.double(var) # 1. Using self
    times_called; var * 2;
  end
  class << self # 2. Using << self
    def times_called
      @@times_called ||= 0; @@times_called += 1
    end
  end
end
def MathFunctions.triple(var) # 3. Outside of class
  times_called; var * 3
end

# No instance created!
puts MathFunctions.double 5 # => 10
puts MathFunctions.triple(3) # => 9
puts MathFunctions.times_called # => 3
```

`self` outside of method refers to `Class` object