In this lecture, we will discuss…

- ✧ Classes
- ✧ How **objects** are **created**
- ✧ How to **access data** within those objects

OO Review

- ✧Identify things your program is **dealing with**
- ✧**Classes** are **things** (*blueprints*)
  - Containers of methods (*behavior*)
- ✧Objects are **instances** of those things
- ✧Objects contain **instance variables** (*state*)

Instance Variables

- ✧ Begin with **@**
  - Example: `@name`
- ✧ Not declared
  - **Spring into existence** when first used
- ✧ **Available to all instance methods** of the class

Object Creation

- ✧ Classes are **factories**
  - Calling `new` method **creates an instance** of class

> `new` causes `initialize`

- ✧ Object's **state** can be (should be) **initialized** inside the `initialize` method, the "constructor"

Object Creation

```ruby
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]
```

Accessing 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

Accessing Data

```ruby
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:
```

Accessing 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…

Accessing Data (Continued)

- Use `attr_*` form instead
  - `attr_accessor` – getter and setter
  - `attr_reader` – getter only
  - `attr_writer` – setter only

Accessing Data (Continued)

```ruby
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
```

Accessing Data (Continued)

- **Two 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 maximum age assigned

Accessing Data (Continued)

Solution: Use a **constructor** and a **more intelligent** age **setter**

But first, we need to talk about `self`…

self

- Inside instance method, `self` refers to the **object itself**
- Usually, using `self` for **calling other methods of the same instance** is **extraneous**

self

- ✧ At other times, using `self` is **required**
  - Ex. - When it could mean a **local variable assignment**
- ✧ Outside instance method definition, `self` refers to the **class itself**

self

```ruby
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?

Summary

- ✧ Objects are **created** with `new`
- ✧ Use the **short form** for setting/getting data (`attr_`)
- ✧ **Don’t forget** `self` when required

What’s next?

- ✧ Class inheritance and class methods