**Attribute Methods**

Let's learn some new methods that can clean up the way we write our classes. So far in the course, we've had to manually implement getter and setter methods for our attributes. Below is a class with some classic getter/setter methods that we'll refactor. Using what we learned so far in the course:

class Dog

def initialize(name, age, favorite\_food)

@name = name

@age = age

@favorite\_food = favorite\_food

end

# getters

def name

@name

end

def age

@age

end

# setters

def name=(new\_name)

@name = new\_name

end

def age=(new\_age)

@age = new\_age

end

end

**attr\_reader**

Creating getter methods for the attributes we want to expose is pretty repetitive. Let's use a new method attr\_reader, to create the getters for name and age automatically:

class Dog

# attr\_reader will define #name and #age getters for us

attr\_reader :name, :age

def initialize(name, age, favorite\_food)

@name = name

@age = age

@favorite\_food = favorite\_food

end

end

dog = Dog.new("Fido", 3, "pizza")

dog.name

dog.age

dog.favorite\_food # NoMethodError: undefined method `favorite\_food', because we didn't pass it to attr\_reader

Let's break down the new line above. attr\_reader is a built-in ruby method that we are calling inside of the Dog class. Note that we don't call it within initialize. attr\_reader should be passed symbols that correspond to the names of the attributes we want to have getters for.

In other words, when we have this call to attr\_reader:

class MyClass

attr\_reader :attribute\_1

# ...

end

It would result in this getter method being defined under the hood:

class MyClass

# ...

def attribute\_1

@attribute\_1

end

end

Don't be thrown off by the syntax we use to call attr\_reader. By convention in Ruby, we omit the parentheses for attr methods. However, attr\_reader :name, :age is equivalent to the explicit attr\_reader(:name, :age).

**attr\_writer**

In a similar way, we can use attr\_writer to define setter methods:

class Dog

# attr\_writer will define #name= and #age= setters for us

attr\_writer :name, :age

def initialize(name, age, favorite\_food)

@name = name

@age = age

@favorite\_food = favorite\_food

end

end

dog = Dog.new("Fido", 3, "pizza")

dog.name = "Spot"

dog.age += 1

p dog #<Dog:0x007fd87f1144a0 @age=4, @favorite\_food="pizza", @name="Spot">

dog.favorite\_food = "calzone" # NoMethodError: undefined method `favorite\_food=', because we didn't pass it to attr\_writer

In other words, when we have this call to attr\_writer:

class MyClass

attr\_writer :attribute\_1

# ...

end

It would result in this setter method being defined under the hood:

class MyClass

# ...

def attribute\_1=(new\_val)

@attribute\_1 = new\_val

end

end

**attr\_accessor**

Oftentimes, we may want both a getter and a setter for an attribute. If we are in this scenario, we can use the attr\_accessor method. It is a combination of attr\_reader and attr\_writer in that it will create both getters and setters for the specified attributes.

class Dog

# attr\_accessor will define #name, #name=, #age, #age= methods for us

attr\_accessor :name, :age

def initialize(name, age, favorite\_food)

@name = name

@age = age

@favorite\_food = favorite\_food

end

end

dog = Dog.new("Fido", 3, "pizza")

# Let's use the setter and getter for name!

dog.name = "Spot"

p dog.name # "Spot"

**Wrapping Up**

Awesome, our code looks much cleaner! However, be cautious. Like we explored in our chat about *encapsulation*, don't just take all your class's attributes and pass them to attr\_accessor. Consider if a user of the class *needs* to manipulate that data with a raw getter or setter. Or more importantly consider if it is *safe* for a user to do so. Only use getters and setters for what you want to expose in your classes.