**Class Basics**

**Motivation**

Before we learn about classes, let's get motivated! Why should we use classes and what advantages do they offer?

**How to Create Cats Poorly**

As a programmer, you'll often want to model some object and the properties of that object. For example, a social media site may need to model a User with their username and a profile picture. Or perhaps a music site may need to model a Song with its title, genre, and duration. Following App Academy tradition, let's say we wanted to model some Cats in ruby! Our Cats will have names, colors, and ages:

cat\_1 = {name: "Sennacy", color: "brown", age: 3}

cat\_2 = {name: "Whiskers", color: "white", age: 5}

cat\_3 = {name: "Garfield", color: "orange", age: 7}

Above we used hashes to represent our Cats. This seems like a fair choice, because we can use the key:value pairs of hashes to represent the properties of our Cats. For example cat\_1 has a name of "Sennacy" and an color of "brown". Now, imagine we wanted to create a thousand Cats. We would have to tediously create each hash with the same keys of name, color, and age. This leaves a lot of opportunity for typos. We want to follow the DRY rule (**D**on't **R**epeat **Y**ourself) and improve this code. By using a class we can avoid this repetition and easily create objects of the same structure.

**Creating a Cat Class**

In the cat example above, we can notice a theme across all Cats we create. They all have the same keys, but may differ in their values. We can say that each Cat has the same structure. Let's DRY up the last example by creating a Class to act as a blueprint for Cats.

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

end

A few things we'll want to note about the code above:

* to create a class we use the class keyword, big surprise
* the name of a class must begin with a capital letter
* we can define methods within a class

You'll notice that we defined a method named initialize in our class. This is a special method name that we will use when creating cats. The method expects 3 parameters, which is nothing new, but what are @name, @color, etc.? @ is how we denote a *instance variable* or *attribute* of our class. That means that our cats will have the attributes or properties of @name, @color, @age.

**Initializing New Cats**

Now that we have a Cat class, we have a blueprint that can easily create Cats for us. Let's put it to use:

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

end

cat\_1 = Cat.new("Sennacy", "brown", 3)

cat\_2 = Cat.new("Whiskers", "white", 5)

p cat\_1 #<Cat:0x007fb6d804cfe0 @age=3, @color="brown", @name="Sennacy">

p cat\_2 #<Cat:0x007fb6d6bb60b8 @age=5, @color="white", @name="Whiskers">

Let's recognize something a bit strange about this code: To create a Cat we must call Cat.new, this must mean that new is a method on Cat. This is strange because nowhere did we define a method literally named new. The trick is, when we call Cat.new, ruby will be really calling upon the initialize method we defined. A hint at this is the fact that the initialize method expects a name, color, age and when we call Cat.new we pass in a name, color, age. You're probably wondering why the heck we can't just call Cat.initialize; it seems way more logical right??? The short answer to that is because reasons. This is something we'll have to accept blindly for now: *Cat.new will execute our initialize method*. As we explore more about classes we promise to explain the weirdness behind new and initialize.

With that out of the way, let's get to the punchline. When we call Cat.new("Sennacy", "brown", 3), it will return an object to us that we store in the variable cat\_1. Notice that the object contains the attributes that result from executing initialize. If we want to create more cats we simply call Cat.new again, passing in any name, color, age we please. We can use our Cat class to create any number of Cat instances. This means that cat\_1 and cat\_2 are instances of Cat.

Notice that when we print out an instance of a class, the notation will show which class this instance belongs to and a unique id for this object: <Cat:0x007fb6d804cfe0...

**Getter Methods**

Since we designed a Cat instance to consist of 3 attributes, it's common to also want a way to refer to the value of those attributes. To do this, we define "Getter Methods" to get (return) those attributes. Let's add a name getter to Cat:

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

def get\_name

@name

end

end

cat\_1 = Cat.new("Sennacy", "brown", 3)

p cat\_1.get\_name # "Sennacy"

Notice that we defined another method called get\_name in our class. To call this method, we must call it on a Cat instance, not directly on the Cat class! This makes sense because cat\_1 is an instance, so it refers to a particular cat. If we had done Cat.get\_name we would be incorrectly trying to get the name of the blueprint. Cat is just the blueprint, so it does not refer to any single, particular cat. In summary we should call cat\_1.get\_name and not Cat.get\_name.

By convention, getter methods typically have the same name as the attribute they are returning. So instead of defining get\_name, we'll simply define name. Let's add another getter using this convention:

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

def name

@name

end

def age

@age

end

end

cat\_1 = Cat.new("Sennacy", "brown", 3)

p cat\_1.name # "Sennacy"

p cat\_1.age # 3

cat\_2 = Cat.new("Whiskers", "white", 5)

p cat\_2.name # "Whiskers"

p cat\_2.age # 5

p cat\_2.color # This will give NoMethodError: undefined method `color'

Cool, so we can now refer to the name and age of any Cat instance! Note that if we don't create a getter for a particular attribute, we won't have a way to refer to that attribute. Such as in the example above, we cannot refer to a Cat's color because we did not create the corresponding getter.

A final thought about getter methods, because they simply return the value of an attribute, we cannot use them to modify the @attribute. So we cannot use a getter method to change a cat's age.

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

def name

@name

end

end

cat\_1 = Cat.new("Sennacy", "brown", 3)

p cat\_1.name # "Sennacy"

cat\_1.name = "Kitty" # This will give NoMethodError: undefined method `name='

To accomplish this behavior we'll need to learn about setter methods next!

**Setter Methods**

Let's see what happens when we try to assign to an attribute of a Cat instance without the proper method in place. The following code will not work:

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

# getter

def age

@age

end

end

cat\_1 = Cat.new("Sennacy", "brown", 3)

cat\_1.age = 42 # NoMethodError: undefined method `age='

The error we get above suggests that we need to have a age= method on our Cat class. What a strange method name! Let's implement it:

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

# getter

def age

@age

end

# setter

def age=(number)

@age = number

end

end

cat\_1 = Cat.new("Sennacy", "brown", 3)

p cat\_1 #<Cat:0x007f8511a6f340 @age=3, @color="brown", @name="Sennacy">

cat\_1.age = 42

p cat\_1 #<Cat:0x007f8511a6f340 @age=42, @color="brown", @name="Sennacy">

Now we have a working method that we can use to change the age! Great. But something that feels uncomfortable here is how we call the method with cat\_1.age = 42. If age= is the method name, then what's up with the space between age and =, as well as the lack of parentheses around our 42 arg? This doesn't seem like a proper method call, but it truly is! The following two method calls are equivalent

cat\_1.age=(42)

cat\_1.age = 42

For setter methods especially, we'll prefer the second version because the syntax is cleaner. Ruby is a quite flexible language. In general you are not required to use parentheses around arguments when making a method call. Try it for yourself: "aeiou".include?("e") is equivalent to "aeiou".include? "e". As a matter of style and convention, we'll only omit parentheses for method calls that don't take in args or are special exceptions like a classic setter method.

**Beyond Getters and Setters**

Getters and setters are common methods to implement on a class, but we can implement any arbitrary method we please on a class. The possibilities are endless:

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

def purr

if @age > 5

puts @name.upcase + " goes purrrrrr..."

else

puts "..."

end

end

end

cat\_1 = Cat.new("Sennacy", "brown", 10)

cat\_1.purr # "SENNACY goes purrrrrr..."

cat\_2 = Cat.new("Whiskers", "white", 3)

cat\_2.purr # "..."

<http://www.sennacy.com/>