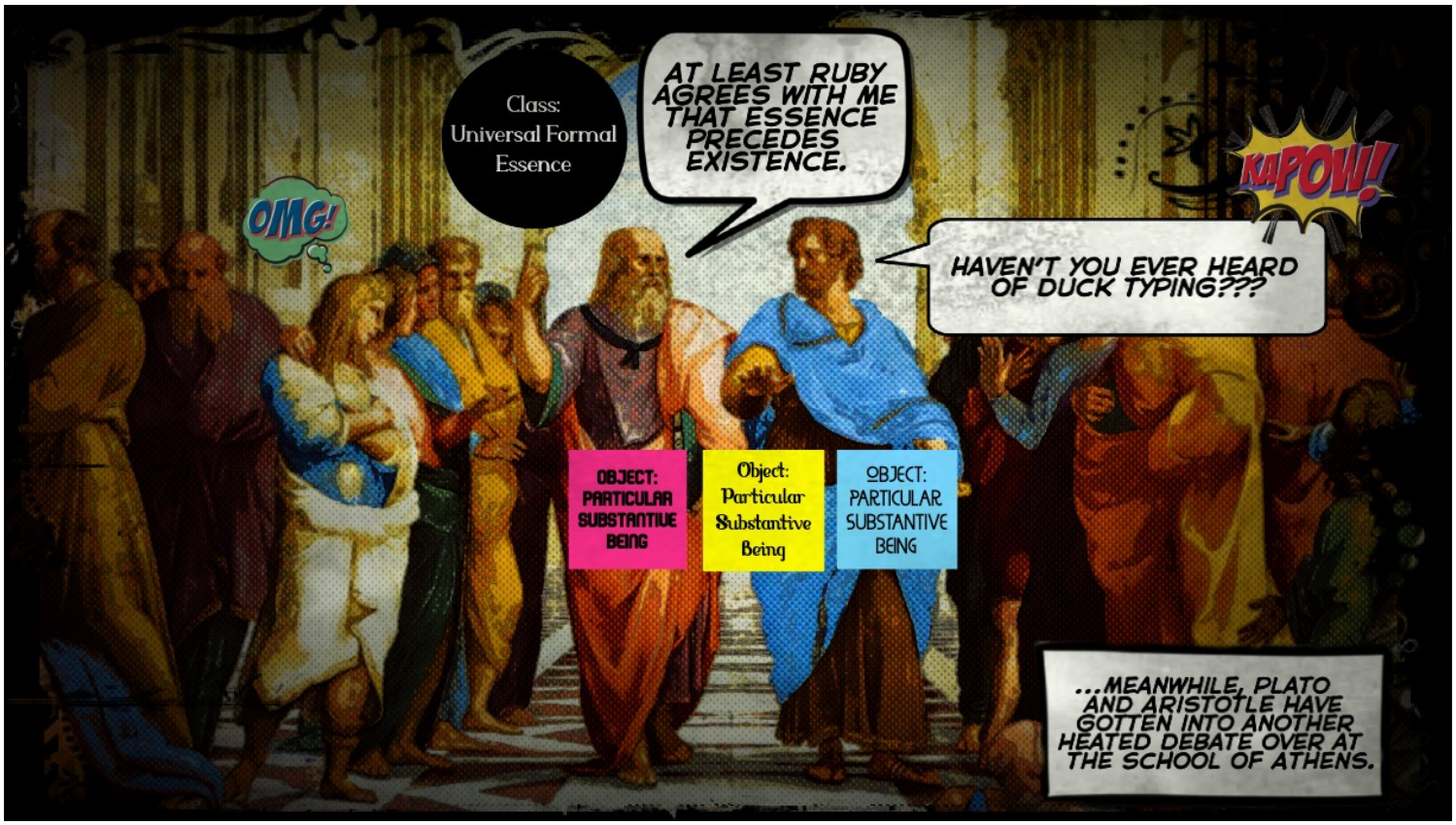


Towards a Conceptual Model of Object Oriented Programming

Matt Johnston Following
Jan 20, 2020 · 19 min read

Ruby and the Problem of "the One and the Many"

How is that we are able to use a term like "dog" to refer to so many different beings in the world? We've got chihuahuas, pugs, poodles, golden retrievers, mastiffs, and the list goes on. Is there some universal essence that allows us to group certain particular beings together into one distinct

category?

This problem of *universals* and *particulars*, also known as the problem of "the One and the Many", is not a new one. Plato was asking the same question thousands of years ago. He came up with a whole theory about it, a theory we refer to as Plato's "Theory of Forms".

Plato, of course, wouldn't have the last word on the matter, and today, it's not just philosophers that grapple with the problem, but computer programmers as well. This is clearly evident for anyone acquainted with Object Oriented Programming (OOP). We're going to discuss the OOP paradigm within the context of the Ruby programming language, focusing especially on how to think and talk about Ruby classes and Ruby objects.

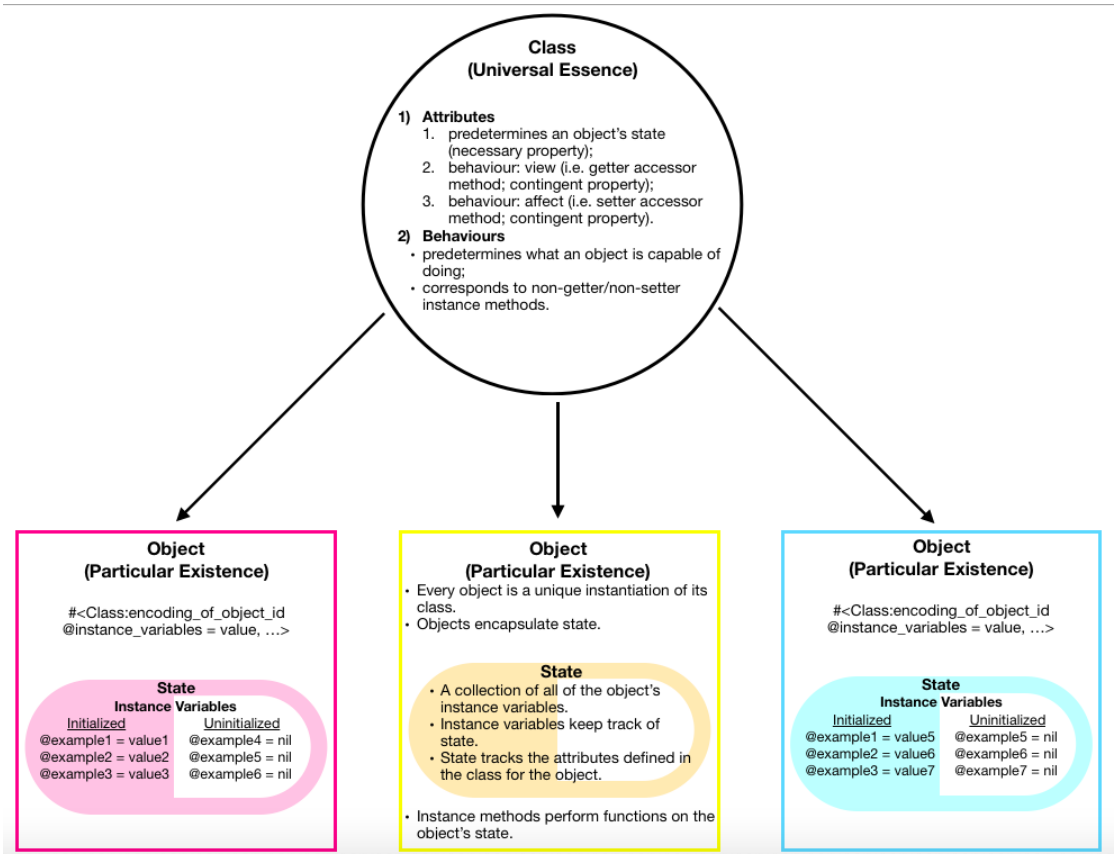
Consider this post as an initial attempt by a newcomer to the programming world at trying to conceptualize the relationship between classes and objects. It's not so much an attempt to "pin things down" or to "have the last word" as it is an attempt to encourage discussion; so critiques and suggestions are welcome. (By the way, the discussion below ignores Ruby's meta-programming features).

First Principles

Let's begin by thinking of Ruby classes as universal forms and Ruby objects as particular beings. A class defines an essence that predetermines or prefigures the particular objects instantiated from that class; more specifically, a class contains defined *behaviours* and *attributes*[1] that govern what any particular object is capable of doing and the purview of any particular object's *state* (the italicized words are emphasized because they are semantically loaded terms within the OOP paradigm, or at least Ruby's implementation of it).

Both behaviours and attributes are defined within a class, but before any object is created, or instantiated, the behaviours and attributes exist only *in potentia*; that is, they exist as formal essential possibilities rather than as substantive particular realities. And in this case, contra the existentialists, *essence precedes existence*.

Below is a visual schema of the relationship between classes and objects. Have a look, but don't get too bogged down by the details just yet. The rest of the discussion will be concerned with explaining this schema in more detail. Refer back to it as desired.



Classes and Objects:

Class (Formal Essence)

As previously mentioned, attributes and behaviours are defined within a class. The combination of these attributes and behaviours comprise the universal essence inhering to every instance, or object, of the class. Just as we might assume that there must be some universal properties inhering within every chihuahua, poodle, and pug that enables us to refer to each one of them as a dog, a class is what defines that essence within the context of the Ruby programming language.

Let's start with an example. Below, we define a `Robot` class, in which we outline the essence of what it is to be a Robot object. In the simple case outlined below, a Robot object has the ability to 'talk' (behaviour) and has a

'name' attribute.

```
# Class Definition -- Formal Essence

class Robot
  def initialize(name)
    @name = name        # Here, we define a 'name' attribute within the class.
  end

  def talk              # Here, we define a 'talk' behaviour within the class.
    puts "I'm a robot, and I can talk."
  end
end
```

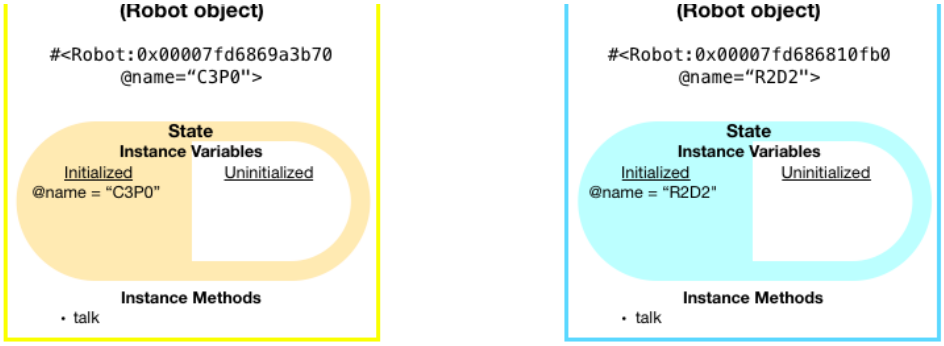
The attribute and behaviour we have defined above within the `Robot` class work to make up the universal essence of every Robot object. Every Robot object instantiated from the `Robot` class will be unique, but we know it's a Robot object because of this universal essence. Currently, that essence consists of a 'name' attribute, which will predetermine that every object of the `Robot` class has an `@name` instance variable, and a 'talk' behaviour, which predetermines that every object of the `Robot` class will have access to a `talk` instance method.

Object (Substantive Existence)

An object, or instance, is a particular instantiation of a class. For example, we can instantiate one, or any number of Robot objects from our `Robot` class. Below, we instantiate two.

```
# Object Instantiation -- Particular Existence

r2d2 = Robot.new("R2D2")
c3p0 = Robot.new("C3P0")
```

Notice that because we defined the `initialize` constructor method with a `name` parameter we must pass an argument to the `new` method when we call it on our `Robot` class in order to instantiate a new Robot object.



Each of the Robot objects instantiated above are unique, each having its own unique object id, which we can reveal by calling the `object_id` method on each one of our objects.

```
r2d2.object_id
# => 70218315657300
c3p0.object_id
# => 70218315588860
```

Upon instantiation, each object also generates its own unique state. By state, we mean the collection of all the instance variables belonging to the object. In the case of our Robot objects above, each object possesses an instance of the 'name' attribute; that is, the instance variable, `@name`. Thus, the collection of instance variables (i.e. state) for each of our Robot objects is accounted for by each object's respective `@name` instance variable.

Instance variables keep track of an object's state. More precisely, instance variables keep track of information about an object's state.[2] For example, the `@name` instance variable belonging to our `r2d2` Robot object references the value `"R2D2"`. This piece of data or information is what comprises the state of an object, and is what the `@name` instance variable is tracking. It is unique from the values associated with the `@name` instance variable belonging to our other Robot object, `c3p0`.

```
p r2d2
# => #<Robot:0x00007fb9f385e0a8 @name="R2D2">
p c3p0
# => #<Robot:0x00007fb9f383c9f8 @name="C3P0">
```

Above, we've called the `p` method on both of our Robot objects, which returns a value containing the name of the class, an encoding of the object id, and the value associated with each object's respective `@name` instance variable. As we can see, each object's respective `@name` instance variable is associated with a different value, evidence that each object has its own unique state.

Indeed, we can even say that objects encapsulate state. For example, we currently have no direct access to the `@name` instance variable and the value it references for any of our Robot objects. We can neither view it nor manipulate it. It is untouchable, or private. We might try to access it and view it by doing something like the following…

```
r2d2.name
```

…but instead of the `r2d2` object allowing the value referenced by its `@name` instance variable to be viewed, we get the following error…

```
# => NoMethodError: undefined method `name' for #<Robot:0x00007ff7900f5c28 @name="R2D2">
```

Hmmmm….an undefined method error. Now might be a good time to turn to discussing behaviour and attributes, which are defined within a class and which comprise the formal essence of any particular object.

Behaviour, Attributes, and their Corresponding Instances

Behaviour and Instance Methods

Behaviours defined in a Ruby class predetermine what any individual object of that class is capable of doing. It's important here to realize that any particular object in question is not strictly limited to the behaviours of its class; it may also perform behaviours defined in any of the classes within its inheritance hierarchy as well as any mixed-in modules.

If we invoke the `ancestors` method on our `Robot` class, an array, whose elements consist of the various classes and modules that comprise the

method lookup path of our `Robot` class, will be returned.

```
Robot.ancestors
# => [Robot, Object, Kernel, BasicObject]
```

This method lookup path is the path our program will traverse when looking for any instance method called on an object of the `Robot` class. The `Robot` class, being the first in the list, will be searched first, followed by the `Object` class, the `Kernel` module, and finally, the `BasicObject` class. If a behaviour is defined in any of the classes/modules along that path, then the instance method corresponding to that behaviour will be accessible to objects of the `Robot` class. In this sense, we might think of instance methods as *instances* of the behaviours defined in either the class or somewhere within its class inheritance hierarchy.

Let's call the `talk` instance method on our Robot objects in order to view the 'talk' behaviour defined in our `Robot` class.

```
r2d2.talk
# => I'm a robot, and I can talk.
c3p0.talk
# => I'm a robot, and I can talk.
```

Invoking the `talk` method on each of the two distinct Robot objects outputs the same message to the screen: `I'm a robot, and I can talk.`. The message itself is irrelevant to our discussion, and indeed, we could have defined the 'talk' behaviour to do any number of things. The important part here is to realize that each of our Robot objects is only capable of performing the specified behaviour when the `talk` instance method is invoked because we have already defined that behaviour within our `Robot` class — essence precedes existence.

That's enough about behaviour for now. We will continue to talk about behaviours throughout the rest of the discussion but let's turn to the other aspect of essence as defined by class — attributes.

Attributes and Instance Variables

Attributes have one *necessary* property and two *contingent* properties. You cannot define an attribute without the necessary property being inherent to that definition, but you can define an attribute without the two contingent properties.

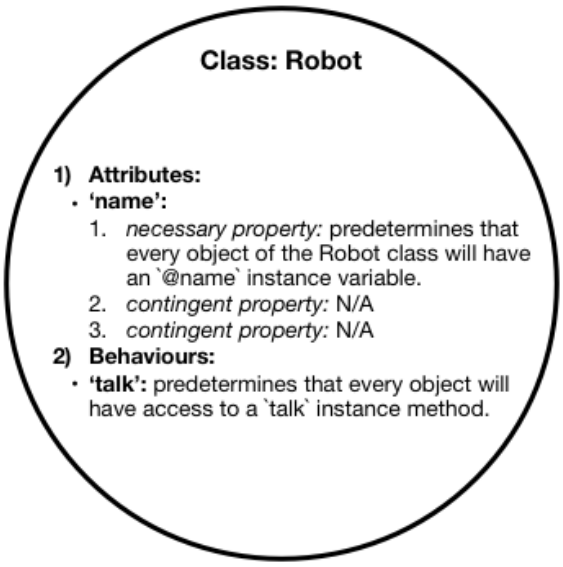
The Necessary Property

The necessary property of any attribute is the function of predetermining the state of any particular object. Every object's state derives its universal form or essence from the attributes defined in the object's class and any classes within the inheritance hierarchy. Thus, an object's state must, and indeed does, track the attributes defined in the object's class and class inheritance hierarchy.[3] This means that an object can only possess instance variables that have been predetermined by the attributes defined within its respective class and inheritance hierarchy.

There can be no *instance* of an attribute without that attribute already being defined — essence precedes existence. This suggests that we should think of instance variables as *instances* of attributes defined in the class, similar to how we can view instance methods as *instances* of behaviours defined in the class, and of course, instances (or objects) are just instances of the class itself.

This leads to a very important and perhaps controversial point concerning ostensible variables whose names start with `@`. For example, in our `Robot` class definition above, we have `@name`. It would appear that we are dealing with an instance variable, but appearances can be deceiving. Here's an important point that David Flanagan and Yukihiro Matsumoto make about instance variables in their book *The Ruby Programming Language*: "All Ruby objects have a set of instance variables. These are not defined by the object's class — they are simply created when a value is assigned to them."[4] Classes don't define instance variables, they define attributes. Instance variables do not exist prior to an object being created and a value being assigned to them.

Top highlight



The distinction between attributes and instance variables is important because it can be tempting to use the term 'attribute' and 'instance variable' interchangeably. This is fine so long as one maintains the distinction between what a class is and how it functions and what an object is and how it functions. But if we are looking to avoid ambiguity, it might be best to speak of `@name` as an instance variable within the context of its belonging to a particular object, and to speak of `@name` as an *attribute signifier* within the context of class definition. Within our `Robot` class definition, the presence of an `@name` attribute signifier will predetermine the existence of an `@name` instance variable inhering to every object of the `Robot` class. Evidently, context matters.

The fact that context matters really shouldn't be all that controversial. The Ruby programming language itself has no absolute sovereign authority over the use of the `@` symbol. In the English language, at least, it's a substitute for "at" in certain contexts, and it's used as a prefix for Instagram handles by people who have no clue what an instance variable is (imagine we started to call them Instagram instance variables). Thus, to speak of `@name` as an attribute signifier within the context of class definition and as an instance variable within the context of any particular object of the class shouldn't be that problematic. Of course, "attribute signifier" does feel bulky.

The Contingent Properties

The two contingent properties of attributes are both behaviours, which means that there is some overlap between attributes and behaviours (we will explain this in more depth further below). These two contingent properties correspond to the two types of Ruby accessor methods: 1) getter methods and 2) setter methods. To illustrate what we mean by getter and

setter methods, let's define behaviours within our `Robot` class that will correspond to each.

```
class Robot
  # ... rest of code omitted for brevity

  def name  # Here, we define a behaviour corresponding to a getter method.
    @name
  end

  def name=(name)  # Here, we define a behaviour corresponding to a setter method.
    @name = name
  end
end
```

Both the getter and setter method defined above look like behaviours, and as already mentioned above, they are. But they are also attributes; or more specifically, they are contingent properties of our 'name' attribute. Each property is contingent, and although we can define either one without an attribute already being defined, once either one of them is defined, the attribute is necessarily defined.

For example, let's completely redefine our `Robot` class without the `initialize` constructor method, but with a behaviour corresponding to a `name` getter method.

```
class Robot
  def name
    @name     # Here, we define a 'name' attribute with a contingent, behavioural property.
  end

  def talk
    puts "I'm a robot, and I can talk."
  end
end
```

Again, it may appear that we have simply defined two behaviours within our class. But the first behaviour, the `name` getter-method definition, is also an attribute, because it contains a line of code that satisfies the necessary property of any attribute. The state of any object of the `Robot` class will be predetermined by the 'name' attribute defined therein, so that the totality of the state of any Robot object will consist of an `@name` instance variable.

What is special about this ‘name’ attribute that distinguishes it from the ‘name’ attribute we defined in our first definition of the `Robot` class, is that this 'name' attribute has the behavioural property of being able to be publicly viewed. Remember that when we tried to view the value associated with our `@name` instance variable for the `r2d2` Robot object by appending that object with `name` above, an "undefined method" error message was returned. But with a `name` getter method, we are able to directly access the object's `@name` instance variable and view the value associated with it.

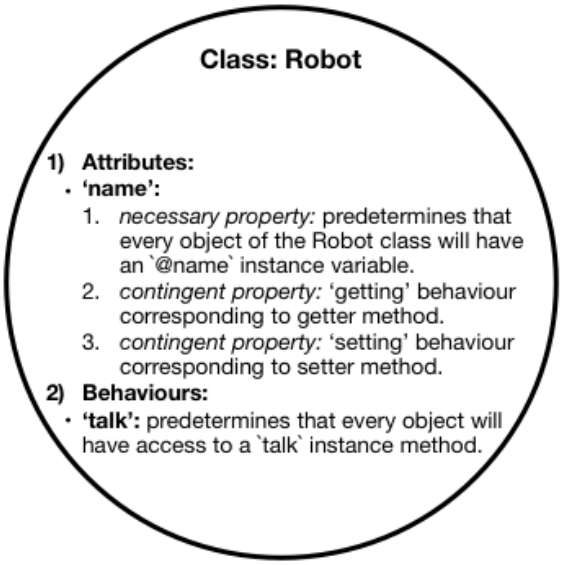
Similarly, if we were to add a `name` setter-method definition to the definition of our `Robot` class, we would also be able to access any Robot object's `@name` instance variable and change its value. Let's add a setter- and getter-method definition to our initial `Robot` class in order to play around with a few examples.

```
class Robot
  def initialize(name) # Constructor method: invoked whenever we instantiate a new object.
    @name = name      # Here, we define an attribute signifier for a 'name' attribute.
  end

  def name    # Here, we are adding a 'getting' behaviour to our 'name' attribute.
    @name     # The 'getting' behaviour is a contingent property of the attribute.
  end

  def name=(name) # Here, we are adding a 'setting' behaviour to our 'name' attribute.
    @name = name  # The 'setting' behaviour is a contingent property of the attribute.
  end

  def talk    # Here, we define a 'talk' behaviour within the class.
    puts "I'm a robot, and I can talk."
  end
end
```

Let’s test them out with one of our Robot objects.

```
c3p0.name
# => "C3P0"
c3p0.name = "Buzz"
c3p0.name
# => "Buzz"
```

Success! We are able to both view and change the value associated with the `c3p0` object's `@name` instance variable.



Getter and setter methods are so common that Ruby has a built-in way of defining them within the class. We can use `attr_reader` to define a getter method, `attr_writer` to define a setter method, or `attr_accessor` to simultaneously define a getter and setter method. Each of these methods take a symbol as an argument. Here's what our code would look like if we replaced our `name` getter-method and `name=` setter-method definitions with a single `attr_accessor`.

```
class Robot
  attr_accessor :name

  def initialize(name) # Constructor method: invoked whenever we instantiate a new object.
    @name = name      # Here, we define an attribute signifier for a 'name' attribute.
  end

  def talk    # Here, we define a 'talk' behaviour within the class.
    puts "I'm a robot, and I can talk."
  end
end
```

And we haven't lost any functionality…

```
c3p0.name
# => "C3P0"
c3p0.name = "Buzz"
c3p0.name
# => "Buzz"
```

…as we are able to view and change the value associated with our `c3p0` object's `@name` instance variable.

But we need to reiterate why it is that we are treating these two behaviours as contingent properties of our 'name' attribute and not just independent

behaviours. The reason has to do with the Object Oriented Programming paradigm and one of its primary goals — *encapsulation*.

Encapsulation

It might help to think about Ruby local variables here. Suppose we just initialize a local variable with the name `robot_name` and assign it to the value `"Buzz"`.

```
robot_name = "Buzz"
```

Notice what happens when we try to return the value referenced by this local variable.

```
robot_name
# => "Buzz"
```

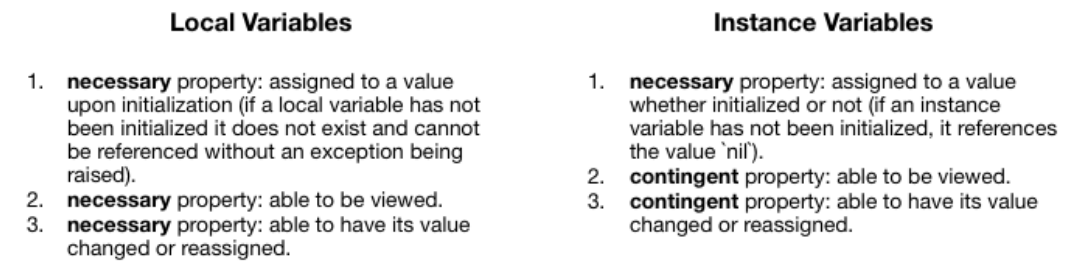
We get what we wanted. What about if we try to change the value associated with the `robot_name` local variable by reassigning it with a new value?

```
robot_name = "Fuzz"
robot_name
# => "Fuzz"
```

Again, we get just what we wanted. But notice how we were able to access our local variable, view it and change it, and all without giving these processes a second thought. It's as if we take these behavioural properties of being able to view and change the local variable's value for granted, as if those properties are inherent to the very essence of being a local variable. This is not the case for instance variables. The flexibility that belongs to instance variables, allowing them to preclude such behaviours, is precisely the precondition for encapsulation.

Encapsulation, a key concept of the OOP paradigm, is a form of data protection that allows programmers to hide functionality from the rest of

the code base.[5] When we define an attribute within a class, we have the flexibility to decide what sorts of properties will belong to it. Once decided, those properties will predetermine the level of access available to the rest of the code base outside of the class; that is, whether the values associated with instance variables are able to be viewed or changed will be predetermined by how we define the attributes within the class.



OOP gives programmers the flexibility to encapsulate state within the object, allowing them to predetermine the accessibility of instance variables. The same flexibility does not pertain to local variables.

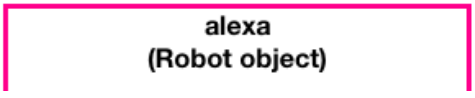
A Note on State and Attribute Tracking

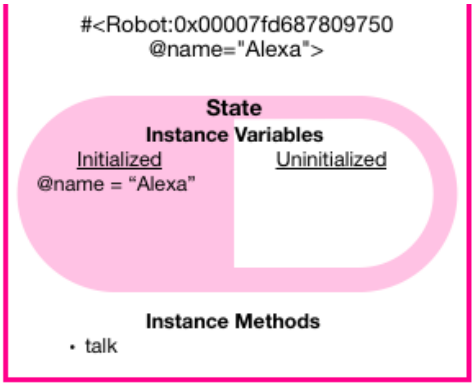
We claimed above that an object's state tracks the attributes defined within the object's class. We want to explain that idea in a little more depth here. Let's use our last definition of the `Robot` class.

```
class Robot
  attr_accessor :name

  def initialize(name)
    @name = name
  end

  def talk
    puts "I'm a robot, and I can talk."
  end
end
```

Now let's create a new instance, or object, of the `Robot` class.

```
alexa = Robot.new("Alexa")
```





If we invoke the `p` method on our `alexa` Robot object, the following is returned...

```
p alexa
# => #<Robot:0x00007f9c5d0f6198 @name="Alexa">
```

We can see that the `alexa` object has an `@name` instance variable associated with the value `"Alexa"`. That instance variable is keeping track of the object's state, but the object's state is tracking the attributes defined in the class for the object. Let's explain.

Suppose that we wanted all Robot objects of the `Robot` class to have the potential to posses musical abilities. Without performing any action on individual Robot objects themselves, we can simply define a 'musical' attribute within our `Robot` class and the state of every existing instance, or object, of the `Robot` class, will now be tracking this attribute. Let's add it, and we will make sure to give it the two contingent behavioural properties of being able to be viewed and changed.

```
class Robot
  attr_accessor :name, :musical  # We add a 'musical' attribute to our class definition.

  def initialize(name)
    @name = name
  end

  def talk
    puts "I'm a robot, and I can talk."
  end
end
```

Without instantiating any new Robot object, let's see what happens when we call the `p` method again on our `alexa` object.

```
p alexa
# => #<Robot:0x00007f9c5d0f6198 @name="Alexa">
```

Hmmm…at first glance, it would seem that our object does not possess an `@musical` instance variable that would correspond to the newly defined 'musical' attribute in our class definition. But that doesn't mean the object's state is not aware of the newly added 'musical' attribute. Indeed, it is aware, which is why we can run the following bit of code without raising an exception, or error.

```
alexa.musical
# => nil
```

The `nil` return value may seem uninteresting since we have not yet assigned a value to any `@musical` instance variable. However, this little example has tremendous significance, for it demonstrates that somehow, something pertaining to our `alexa` object is aware that something like a 'musical' attribute exists. That something is the object's state. The state is keeping track of all the attributes defined within the class. If it wasn't, we should expect `alexa.musical` to return an error message instead of `nil`, just as an uninitialized local variable would if we tried to reference it.

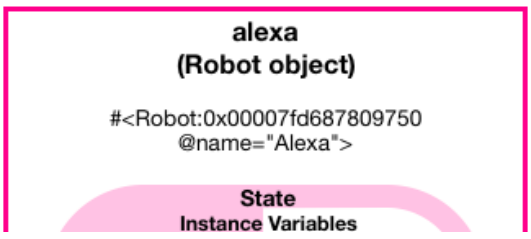
This example suggests another important point that needs to be made. The reason that our invocation of the `p` method on `alexa` did not produce any indication of a 'musical' attribute nor a corresponding `@musical` instance variable, is that the `@musical` variable is as of yet, uninitialized. All uninitialized instance variables reference `nil`. The

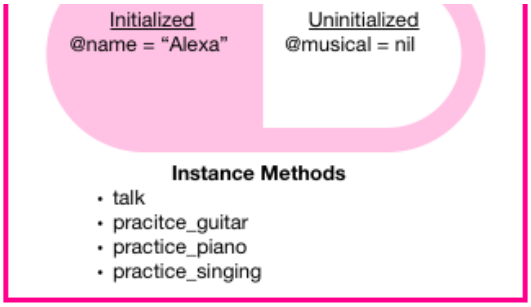
`@musical` instance variable has a certain shadowy existence at this point. It is not non-existent, but its existence is a kind of null existence, or `nil` existence.

To bring the example to life a little bit, let's suppose that we have a robot named Alexa and Alexa has been created with the potential for musical abilities. Currently, Alexa does not play any instrument, nor does she sing, but she has latent musical abilities that exist *in potentia*. Her musical abilities do not have a positive existence. Her musical abilities are not nothing, as they would be say for a rock, but they do have a kind of null, or nil, existence (to speak of the musical abilities of a rock would be to speak nonsense). Even in the world of numbers, zero is still a value nonetheless.

But now let's say Alexa starts practicing the guitar. All of a sudden, Alexa's musical abilities have a positive existence — she now has musical abilities. It's as if practicing the guitar *initialized* Alexa's musical potential and brought it out of its nil existential state, making it a substantive reality rather than just a formal possibility. If we were to try to mimic this example with Ruby, we could define a few 'practice' behaviours pertaining to different musical instruments within our `Robot` class.

```ruby
class Robot
  # ... rest of code omitted for brevity

  def practice_guitar
    musical ? self.musical << 'I play guitar' : self.musical = ['I play guitar']
  end

  def practice_piano
    musical ? self.musical << 'I play piano' : self.musical = ['I play piano']
  end

  def practice_singing
    musical ? self.musical << 'I sing' : self.musical = ['I sing']
  end
end
```



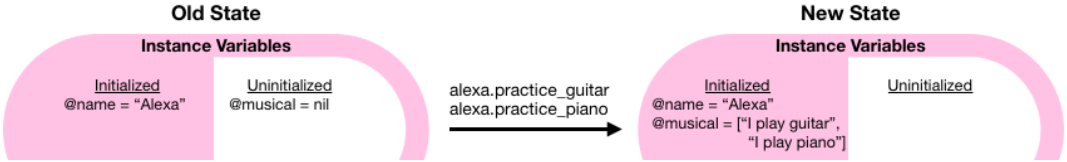


Let's initialize the musical abilities of `alexa` by invoking the `practice_guitar` and `practice_piano` methods.

```
alexa.practice_guitar
alexa.practice_piano
puts alexa.musical
# => I play guitar
# => I play piano
```

The state of `alexa` has been transformed as the `@musical` instance variable has been initialized and assigned with new values. Notice that the invocation of the `practice_guitar` method *initializes* the `@musical` instance variable; the call to the `musical` getter method in the ternary operator returns the `nil` value associated with the `@musical` instance variable, and this *falsey* value causes the ternary operator to evaluate to `false`, which in turn leads to the `self.musical = ['I play guitar']` section of code to the right of the `:` operator being executed instead of the `self.musical << 'I play guitar'` section of code.

However, since the `@musical` instance variable has already been initialized when we invoke the `practice_piano` method on `alexa`, the `musical` getter method returns `["guitar"]`, which is a *truthy* value and thus the ternary operator evaluates to `true`. That causes the `self.musical << 'I play piano'` section of code to be executed instead of the `self.musical = ['I play piano']` section of code.



Instance methods can affect an object's state. Instance variables keep track of the object's state.

The `alexa` object's latent musical abilities have now been realized through practicing the guitar and the piano. She is still the same Robot object, but her state has been transformed — she is the same, but different. We could continue to transform the state of `alexa` by invoking the `practice_singing` method on `alexa`, and the `@musical` instance variable would keep track of this change. Thus, while state tracks attributes for the object, instance variables keep track of an object's state.

Concluding Remarks

To sum up, here is a list of some of the major points that were covered:

1. Classes define an essence for objects, consisting of attributes and behaviours.
2. Objects are instantiated from classes and are predetermined by the class definition.
3. An object's state tracks the attributes of the class, and an object's instance variables keep track of its state.
4. Class behaviours predetermine the instance methods accessible to every particular object of the class.
5. Class attributes predetermine the instance variables pertaining to every particular object of the class.
6. Attributes may posses two contingent behavioural properties, and the contingency of these two properties is a precondition for encapsulation.

Those are the major points. Obviously, there are many things we didn't discuss, such as how to think about class variables and constant variables and whether or not we can neatly file them under the term attribute. The important thing here was to outline as much of the core model around objects and classes without going too deep into the more exceptional cases.

We also didn't say much about the concept of inheritance, aside from

mentioning class inheritance hierarchy. But based on the general model outlined here, we can say that when one class inherits from another, it inherits both attributes and behaviours. Based on the distinction between attributes and instance variables that we have made, it should be evident that classes don't inherit instance variables. Instance variables are unique to their particular objects and do not exist outside of them.

One last point. We've emphasized how, in the OOP paradigm within Ruby, essence precedes existence. There may be a slight exception, however — duck typing. *If it walks like a duck and quacks like a duck, then it must be a duck*. Regardless of what class a particular object is instantiated from, if it is able to behave in a way that is similar to how objects instantiated from other classes behave, then such objects may be able to be treated as a common type for certain applications. In a certain sense then, duck typing is evidence that sometimes, how an object behaves determines what it is. Thus, the existentialists have a point: existence may indeed precede essence after all.

Of course, Plato would argue that even behaviours must be codified prior to any being's ability to perform those behaviours. An object cannot behave in a way that is contrary to its essence; if the behaviour isn't defined in the object's class or class inheritance hierarchy, the object won't be able to behave in that way. But it's nice that even Ruby provides us with some of the dynamism and ambiguity existing in the real world.

The inspiration to write this blog post was prompted by questions that arose while I was studying for a test on OOP at Launch School, an online mastery-based software engineering school, and by a subsequent discussion I had with fellow students and TAs. Thanks to all who contributed to that discussion.

Notes

1. Launch School. *Object Oriented Programming*. "Classes Define Objects". Accessed Jan. 18, 2020.
2. Launch School. *Object Oriented Programming.* "States and Behaviors". Accessed Jan. 18, 2020.
3. Ibid.
4. Flanagan, David and Yukihiro Matsumoto. *The Ruby Programming Language.* California: O'Reilly Media, 2008 (pg. 240).
5. Launch School. *Object Oriented Programming.* "Why Object Oriented Programming?". Accessed Jan. 18, 2020.

Ruby Oop Oop Concepts Launch School Programming