# Ruby Classes and OOP

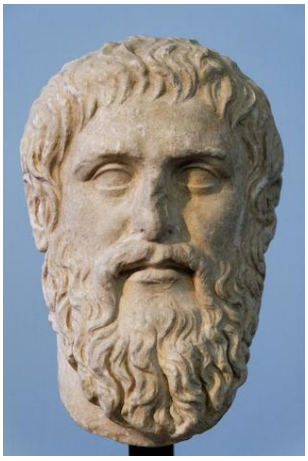
# Prerequisites

Ruby methods & TDD

# What you'll learn

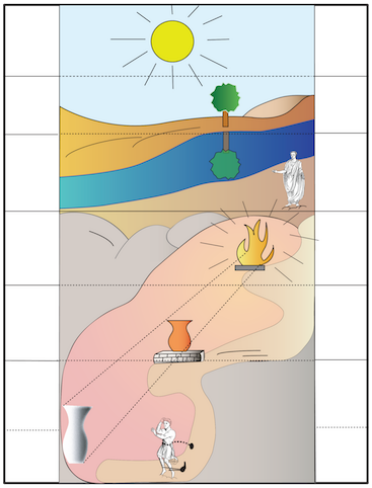
- Object-oriented programming (OOP)
- Classes
- More variable scopes
- Inheritance and Mixins
- OO Thinking

# Theory of Forms



Object-oriented programming isn't so much a programming style as a way of looking at the world - a very very old way. Parts of the idea trace back to Plato some 300-400 years BC. His "Theory of Forms" was inspired by a quest to determine the fundamental nature of reality. Quite literally, he asks "What are things"? "What are objects"?

# Cave Allegory



In the dialogue *The Republic*, Plato presented a thought experiment. Imagine a group of prisoners who have lived their whole lives chained to a wall inside a cave. Their only experience of the outside world is via shadows cast upon the cave walls. For these prisoners, the shadows are reality. They may not realize that the shadows are just reflections of real objects, animals, and people outside the cave.

Plato wonders, “What if this is *really* the way the world works?”

Plato proposes that everything we see - objects, animals, people - are just reflections of a truer reality. There exists some mysterious world, out of reach to our five senses, where *ideas* play and interact. What we experience is only the reflection of that interaction, like the shadows on the cave wall.

A philosopher’s duty, according to Plato is to understand these Platonic Forms (or Ideals). A philosopher’s duty is to escape the cave, behold these ideals, and return to share that knowledge with everyone.

This is just a metaphor, of course. You can only escape the cave in your mind.

# The Class of Tables

Let’s consider, for example, tables. Let’s attempt to grasp the true Platonic Form of a table.

There are many *instances* of a tables in the world. Both of these things are tables. But they are also very different. Yet we still call them “tables”. Why?

There’s some essence of “Tableness” that they both share. Despite their many differences, we can easily come to an agreement that both of these things are “tables”.

So what does it mean to be a “table”? What are the features that come to represent the concept of “Tableness”? What features these two two tables share?

To know the answer to these questions is to gaze upon the Platonic Ideal of “Tableness”.

The term “table” represents an entire *class* of objects. There are an infinite variety of *instances*, but they can all be categorized under the *class* “table”.

# Class

```
class Table

end
```

Let’s create a Table class in Ruby. Just use the keyword “class” followed by the class name, and, as usual, end the block with end.

# Class

```
puts Table.class
print Table.methods

puts Table.hash
```

And, just like that, we've created a brand new Ruby data type.

We can introspect it. It *is* a "Class". It also has a bunch of methods right from the get-go, including, for example, `hash` (which, like I said earlier, is available on every Ruby object).

# Class

```
class Table

end
```

These two short lines of code represent our humble attempt to define the Platonic Form of a Table.

Within this class we will attempt to describe, in Ruby code, all the things that make a table, a table. We will use Ruby to describe the “tableness” that all tables share.

# Instance

```
a_table = Table.new

puts a_table.class
print a_table.methods

puts a_table.hash
```

But before we do that, let's confirm that this Class can in fact create instances of tables. What does this Platonic Form look like when it casts its shadow upon the cave wall?

To generate a particular *instance* of a table from its Class, we use the `new` method. An *instance* of a table is a specific table, just one out of the infinite number of tables that could possibly exist.

We can introspect this creation.

An instance of table has the class `Table`. It belongs to a class of objects called "Table".

A table instance even comes with a list of methods by default. The list includes our friend `hash` - because everything in Ruby can be used as the key in a hash, even this new thing we just created.

# Factory Metaphor

```
a_table = Table.new
```

For some people, the whole philosophical metaphor is a bit too abstract. In that case, you can also think about the relationship between classes and instances as similar to the relationship between a factory and the products it makes.

The `Table Class` is a factory. It contains everything you need to create a table: wood, metal, nails, screws, paint, templates, patterns, workers, and so on.

Calling `new` on the `Table Class` is like asking the table factory to make you a new table. The object that you get back is a single instance of a table.

So a `Class` and an instance of a `Class` are very different things. The `Class` understands how to *make* tables. The instance *is* a table.

As far as the ancient Greeks are concerned, a table factory may as well be Tablos, the god of tables.

Just pick whichever metaphor suits you best. I'm probably going to be switching back-and-forth as we go along.

# Instance Methods

```
class Table
  def put_on(something)
      [] << something
  end

  def look_at
      []
  end
end
```

Like modules, classes can include methods.

A module is used to group a bunch of methods under a single category. That helps you stay organized as you write more and more methods.

A class, on the other hand, is used to define a bunch of methods relevant to a particular concept. For example, adding a method to the `Table` class means defining something we think all tables should be able to *do*, a fundamental aspect of tableness.

What are some methods that we think every table should be able to execute?

I should be able to “put something on” a table. Then, when I later “look at” the table, I should be able to see the thing I just put on it.

So let’s add methods `put_on` and `look_at` our table, with some basic implementations.

# Instance Methods

```
a_table = Table.new
a_table.put_on 1
puts a_table.look_at
```

So let's get a new instance of our table and put a number 1 on it, and look at it.

This doesn't work. When I look at the table, I'm not seeing the number 1.

# Instance Variable

```
def put_on(something)
    tabletop = []
    tabletop << something
end

def look_at
    tabletop
end
```

Classes, unlike Modules, have some internal state. They are state machines. A LightBulb class, for example, should be able to tell me if it's on or off. A Car class should be able to tell me if it's moving or stopped.

To store state we need some variables. Let's add a variable to our methods that keeps track of the items on top of the table.

# Instance Methods

```
a_table = Table.new
a_table.put_on 1
puts a_table.look_at
```

This still isn't working. Now the look_at method is generating an error.

If this were a test (and it should be), this test would be failing.

# Instance Variable

```
def put_on(something)
    tabletop = []
    tabletop << something
end

def look_at
    tabletop
end
```

The variable, tabletop, is scoped to the method. The variable I define in put_on doesn’t exist outside put_on. So when look_at tries to use it, it can’t find it. For an instance of a class to track it’s internal state, we need to define a variable that is scoped to entire instance, not just a method.

# Instance Variable

```
def put_on(something)
    @tabletop = []
    @tabletop << something
end

def look_at
    @tabletop
end
```

Variables that start with the at (@) character are scoped to the instance. So every instance of a Table has it's own tabletop variable.

# Instance Variables

```
a_table = Table.new
a_table.put_on 1
puts a_table.look_at
```

Now our code works. I can put the number 1 on the table, and when I look_at the table I'll get it back.

But we still have a problem.

# Initialize

```
a_table = Table.new
a_table.put_on 1
puts a_table.look_at

a_table.put_on 2
puts a_table.look_at
```

I’m using an Array to track the items on my table. But when I put another item on my table, I clear off the table first. So instead of getting 1 and 2, I just get 2. That’s not how a table works. A table can hold more than 1 item at a time.

# Initialize

```
class Table
    def put_on(something)
        @tabletop = []
        @tabletop << something
    end

    def look_at
        @tabletop
    end
end
```

The problem is the first line in put_on. I reset the tabletop to an empty array before I put anything on the table. What I want to do is initialize the tabletop when the table is created, and only when the table is created.

# Initialize

```
class Table
    def initialize
        @tabletop = []
    end
    def put_on(something)
        @tabletop << something
    end

etc…
```

Ruby classes have a special method for that. The “initialize” method is called when a new Object instance is created from a Class. This is a great place to setup all the instance-scoped variables you’d like to use to track internal state.

Initialize is guaranteed to run every time you call new, so there’s no chance you’ll be able to create a table without a tabletop.

# Initialize

```
a_table = Table.new
a_table.put_on 1
a_table.put_on 2

puts a_table.look_at
```

Now our code works. When I put the number 1 and the number 2 on the table, I can get them back again.

# Initialize

```
class Table
    def initialize
        @tabletop = []
        @num_legs = ?
    end
end
```

Tables also have legs. Maybe it's 4 legs. Maybe it's one big leg. Maybe it's something else. But all tables must have legs. Otherwise it's just a tabletop on the floor. So let's define a variable to track the number of table legs. And let's initialize it in the initialize method. That way we can make sure all of our tables have legs.

But what do we initialize the num_legs variable to?

# Initialize

```
class Table
    def initialize(num_legs)
        @tabletop = []
        @num_legs = num_legs
    end
end
```

What we need is to allow the initialize method to accept arguments. Then we can allow the user of our Table class to let us know how many legs their specific table instance has.

Ruby actually allows this.

# Initialize

~~`a_table = Table.new`~~

wrong number of arguments (0 for 1)

```
a_table = Table.new 4
```

Since we've added an argument to our table initializer, Ruby won't allow tables to be created without that argument. So our old call to Table.new fails with an error letting us know that we now need 1 argument to create a table.

With an argument passed in, we can now successfully create an instance of our table.

# Attributes

```
a_table = Table.new 4

puts a_table.num_legs
```

undefined method `num_legs'

Now it would probably be nice to be able to view how many legs my table has. So let’s try to access that instance variable.

Nope, I can’t get it. It’s looking for a method named “num_legs”, not a variable named “@num_legs”.

# Attributes

```
class Table
    def initialize(num_legs)
        @tabletop = []
        @num_legs = num_legs
    end

    def num_legs
        @num_legs
    end
end
```

So let’s create a method to get my variable back out. It’s simple enough. Just return the instance variable.

# Attributes

```
a_table = Table.new 4
puts a_table.num_legs
```

Great, now my code works. I got my 4 back.

# Attributes

```
class Table
    attr_reader :num_legs

    def initialize(num_legs)
        @contents = []
        @num_legs = num_legs
    end
end
```

This is actually a very common pattern in Object-oriented programming. I have some instance variable that I'd like to make visible to people who use my Class. It's so common, in fact, that Ruby has a shortcut.

The attr_reader method is a shortcut for the 3-line method I wrote earlier. All I need is a symbol naming my variable.

# Attributes

```
a_table = Table.new 4
puts a_table.num_legs
```

And, if I delete the old method, my testing code still works. I can still set the number of legs and get it back.

# Attributes

```
def num_legs=(value)
  @num_legs=value
end
```

Can be written as:

```
attr_writer :num_legs
```

There are other attribute method shortcuts as well.

attr_writer is a one-line shortcut for a method that allows me to reset an instance variable.

# Attributes

```
a_table = Table.new 4
puts a_table.num_legs

a_table.num_legs = 1
puts a_table.num_legs
```

So attr_writer allows me to do this to reset the num_legs variable.

# Attributes

```
attr_reader :num_legs
attr_writer :num_legs
```

Is the same as:

```
attr_accessor :num_legs
```

The attr_accessor shortcut allows me to combine both the attr_reader and the attr_writer shortcuts. That's like 6 lines of code all rolled up into 1.

# Class Scope


What if, as Plato recommended, I want to exit the cave and gaze upon the pure essence of the Platonic form? What if I want to define something about all tables, not just individual tables?

# Class Scope

```
class Table
    def self.has_legs?
        true
    end
end
```

So, acting as the God of Tables, I have declared that all tables, in order to be tables, must have legs. I've defined a method, prefixed with the word self, that tells me the inherent truth of Tables.

This should look familiar. This is how methods are defined within modules. Modules are like classes that can't be instantiated. Modules only have module-level scope. There are no instances.

# Class Scope

```
puts Table.has_legs?
```

Since I've declared a method on the Table class, I don't need to create a Table object to use it. I can just reference it directly from the Table class.

Using the class-wide scope is relatively rare. Playing around with class-scoped objects leads to some very advanced Ruby code. So, if you decide to use it, keep it simple.

More often than not, you'll be using code that defines class-scoped methods, but you won't be defining them yourselves.

# Global Scope

```
$global = true
```

Beyond method scope, beyond class scope, there is the ultimate scope. Globally scoped variables are defined once and are available everywhere. They begin with the dollar sign.

There are very few circumstances that call for such a powerful variable. The problem with globally scoped variables is that you can never be sure who or what is going to come along and change them out from under you. One minute you'll set it to true, then some deep dark code in some file somewhere will flip it to false without even telling you. If we're assuming all users are evil or dumb, including our fellow coders, then global variables are where they can do the most damage. Global variables are the devil's playthings. They begin with a dollar sign because money is the root of all evil.

# Global Variables

`$:`
Where Ruby will look for files you require.

`$0`
The name of the current Ruby file.

`$_`
The last thing read by gets.

That being said, there are certain global variables that Ruby defines for you (more than I have listed here). Some of these variables are blessed and cannot be changed, but not all of them.

There's a special place in hell for people who willingly change the value of these variables.

# Inheritance

```
class Car
end

puts Car.class
print Car.methods

my_car = Car.new
puts my_car.class
print my_car.methods
```

Let's go back a few steps. I've shown you how to create a class and add some methods and variables. But I also showed you that a completely empty class comes with some stuff for free.

So this empty Car class, for example, has a method named "class". An instance of a Car also has a method named "class". In fact, the Car class comes with 95 methods, and a car instance comes with 54 methods. But I haven't defined any of them. So where do they come from?

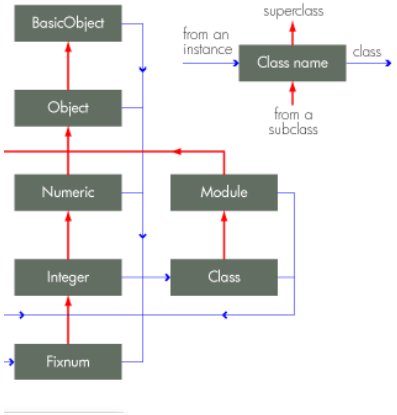
# Inheritance

```
print Car.ancestors
```

[Car, Object, Kernel, BasicObject]

The Car class inherits methods from it's ancestors. An empty Car class, for example, contains methods that have been defined in the Object, Kernel, and BasicObject classes.

# Inheritance

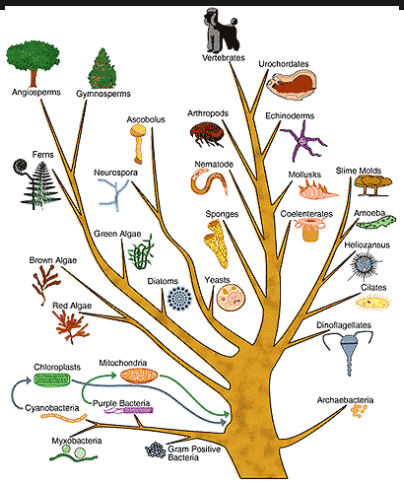



In fact, everything I've ever shown you, all the data types, form a hierarchy. At the top, the root of everything, is the BasicObject. Then comes the Object. Then comes everything else. Numbers, Strings, Arrays, Hashes, Ranges. Even Modules and Classes themselves are a branch in the hierarchy. Everything is an object.

Each new object in the tree, growing from top to bottom, inherits everything above it, everything from it's ancestors, then adds something new.

source: http://skilldrick.co.uk/2011/08/understanding-the-ruby-object-model/

# Inheritance




So, like the tree of life, objects grow more complicated the further your are into the tree. Simple structures grow more advanced with the addition of new methods and variables. Complex designs evolve from simpler ones.

In Ruby, we aren't limited to creating simple nodes in the tree. We can create entire branches.

Source: http://www.all-creatures.org/hope/gw/02_paleobiological_background.htm

# Inheritance

```
class Car
  attr_accessor :engine
  attr_accessor :tires
end

class Motorcycle
  attr_accessor :engine
  attr_accessor :tires
end
```

Let's start with two classes, a Car class and a Motorcycle class. They're both very similar. They both have engines and they both have tires. You can drive both of them. You can park both of them. Very similar.

The first thing you should notice is that this violates the DRY design principle. We're repeating ourselves way too much. This code is too wet. We need to DRY it up. Let's refactor.

# Inheritance

```
class Vehicle
  attr_accessor :engine
  attr_accessor :tires
end

class Car < Vehicle
end

class Motorcycle < Vehicle
end
```

Using inheritance, we can pull the repetitious code into a superclass called Vehicle. Then we can have the Car and Motorcycle classes inherit from Vehicle. We use the less-than sign to tell Ruby that our Class should inherit everything from another class.

This is called a “pull-up refactoring” and it’s very common.

# Inheritance

```
print Car.ancestors
puts Car.superclass

print Motorcycle.ancestors
puts Motorcycle.superclass
```

With our own custom branch of the object hierarchy in place, we can introspect to see that Vehicle is now an ancestor of Car. In fact, Vehicle is the immediate ancestor, the first parent, making it the “superclass” of Car.

The same goes for Motorcycle.

# Inheritance

```
car = Car.new
puts car.engine

motorcycle = Motorcycle.new
puts motorcycle.engine
```

With our shorter refactored code in place, we can now test to make sure that both Motorcycles and Cars have engines. This code doesn't trigger an error, so we're good.

# Modules and Inheritance


Modules don't fit directly into the object hierarchy. Modules represent something called a "cross-cutting concern". That means that modules collect methods that can apply to multiple, unrelated nodes in the object hierarchy.

For example, let's say you have talking car. The ability to talk isn't inherent to the concept of a Car. All cars don't talk. A car doesn't have to talk to be a car. So the "talk" method shouldn't be part of the Car class.

# Modules and Inheritance


But other things can talk. If we just focus on mechanical things, for example, androids can talk. Some toys can talk. But they're not very closely related to cars.

# Mixins

```
module Talkative
  def speak
      puts "Hello world!"
  end
end

class Kitt < Car
    include Talkative
end
```

So let's collect some useful methods for speech in a Talkative module. Then let's create a special subclass of Car for Kitt, the Knight Rider car, and have it include the Talkative module.

Note that, in this case, the method in the module is not being defined with the prefix "self". Recall that include goes one step beyond require. We're essentially copying and pasting the methods in the Talkative module into the Kitt class. The module, in this case, is being used as a "mixin". It's being "mixed into" the Class.

# Mixins

```
print Kitt.ancestors
puts Kitt.superclass

kitt = Kitt.new
kitt.speak
```

We can confirm that Kitt's ancestors now include both Car and the Talkative module. But even Kitt's immediate ancestor, it's superclass, is still Car.

And with this module in place, the kitt car can now speak.

# Why OOP?

Why is Object-Oriented Programming the way to go? Why are so many languages object-oriented? Why base a software design on the philosophical realism of Plato and not the idealism of Kant (pictured here) or even plain nominalism? Why do we depend more on Class inheritance rather than Modules?

It's not a question of taste or philosophical truth. It's a simple fact of engineering. For many types of software problems, object-oriented solutions are just better.

# Thinking Functionally

```
def add(left, right)
  left + right
end
```

Up to now I've asked you to think functionally. We've defined Class-less functions in a plain old Ruby file. But Ruby isn't designed functionally. It's designed objectionally (objectively? object-orientationally?) - whatever it's called.

Take addition, for example. Thinking functionally, we might design addition like this. Add, as a method, exists in a vacuum, without any context. It takes two arguments, a left and right side, and it adds them together. Seems simple enough.

# Thinking Functionally

```
def add(left, right)
  # if Strings, concat
  # else
  left + right
end
```

But think about all the different ways Ruby uses addition. You can add Strings together, for example. String addition uses concatenation. Instead of writing that code, I'm just going to make a note about it.

# Thinking Functionally

```
def add(left, right)
  # if Strings, String.concat
  # if Arrays, Array.concat
  # if nils, error
  # else
  left + right
end
```

What if either of the arguments is nil? In that case we need to trigger an error.

# Thinking Functionally

```
def add(left, right)
  # if String, String.concat
  # if Array, Array.concat
  # if Nil or Hash, error
  # else
  left + right
end
```

Hashes don’t support addition at all, so, like nil, they should have to return errors as well.

# Thinking Functionally

```
def add(left, right)
  # if String, String.concat
  # if Array, Array.concat
  # if Nil or Hash, error
  # If anything that's not a number, error
  # else
  left + right
end
```

What about new Classes, like the Car or Table we created earlier. Those won't work either.

# Thinking Functionally

```
def add(left, right)
  # if String, String.concat
  # if Array, Array.concat
  # if Nil or Hash, error
  # If anything that's not a number, error
  # Unless it's related to a number
  # Unless it knows how to add some other way.
  # else
  left + right
end
```

But what if we create something new that does support addition? Maybe it's a subclass of a number? Or maybe it does something special with addition, like String or Array.

This is getting out of control. We're going about this the wrong way. It's just not possible to define addition functionally in a language that supports the creation of new kinds of data. We would need to create a method that understands all the types of data that ship with the language, plus a bunch of rules for dealing with any new types of data people might create in the future. If a method like this were even possible, it would be doing way too much. With all these lines of code, something is bound to go wrong. It's a bad design.

And this is just addition. What about all the other mathematical operators?

# OOP Thinking

```
class Table
    attr_accessor :tabletop
    def initialize
        @tabletop = []
    end

    def add(item)
        @tabletop.push(item)
    end
end
```

Object-oriented thinking helps us shrink the problem. Instead of having to think about addition in the abstract, we only have to think about it in the context of a particular class.

So let's consider our Table class from earlier. If we want to create an "add" method for tables, we only have to think about what it means to add something to a table. Does our table care what that thing is? In this case, no, it doesn't. The tabletop is just an Array. You can add whatever you want to an array.

# OOP Thinking

```
table = Table.new
table.add(nil)
table.add({})
table.add([])
table.add(Table.new)
puts table.tabletop
```

Once we orient ourselves around an object, in this case the table, then we are free to do all sorts of things. And we don't have to worry about changing what addition means to other objects. This is table's "add" method, not everyone's "add" method.

# OOP Thinking

Functional design:

```
add(something, something)
```

What if something is a Table?

OOP design:

```
object.add(something)
```

Does object care if something is a Table?

So in a functional design, we have to create methods that care about everything. In Object-Oriented design, we can create methods that only care about what the object needs. The less a method has to worry about, the smaller, faster, and better it is. That's the engineering reason for OOP.

# Ruby Syntax

~~subject~~.method(argument, argument)

object.method(argument, argument)

Now, after all that, it's finally time to revisit our basic Ruby syntax. What we had been calling the subject of the Ruby sentence is actually an object. In fact, it's always an object. Because everything is an object. Even if it's blank and assumed to be "self", self is still an object.

The object at the beginning of every line of Ruby is what orients the rest of the Ruby expression. It provides a context for understanding what the method does.

The standard Ruby syntax is object-oriented.