# PROCEDURAL VS OBJECT-ORIENTED PROGRAMMING

- So far, we have done a lot of *procedural* programming.
- Our programs just read from top to bottom, do this, do that, if this is the case, then do something else.
- This gets repetitive, cumbersome and actually dangerous
- For example: Three hashes representing students.

# CLASSES

- They represent a type of thing
- Those things can perform actions
  - Actions are performed with methods you create inside the Class
- The can contain their own values (think variables, but just for the class)

# CLASSES

- Everything in Ruby has a class. Every data type we have touched is an object. Even `nil` is an object with a class!
- Arrays
- Hashes
- Strings
- Even true or false has a class

# CHECKING WHAT A THING IS

- You can always call .class on an object to get its class.
- For example
- `"Im a string".class`
- This would give you `String`

- A class can be thought of like a template, or blue print.
- They contain the basic details of how something should look, but not the final detail
- An object, or instance  of a class is when you build that blueprint.
- Think: Cookie Cutter house blueprints are the class, the actual houses when built, are instances of that blueprint

Class =

the abstract idea of 'house'
There are many like it.

Instances of that class =

This house in particular.
There are many like it but
this one is mine.

- Classes start with:
  - `class MyClassName`
- And end with:
  - end

```
class MyNewClass
end
```

- Classes MUST start with a capital letter

You create Objects from classes by...

```
class House
end

# instantiate a new object from the class
my_house = House.new
```

- You can define one or more attributes on a class by typing:
  - attr_accessor :name, :age, :favorite_food
- An attribute is to contain data within an instance of a class
- Think: When we build a house from a blueprint, the color of paint is different per house. It is an attribute of the house.
- So paint color would be an attribute of the house

# GETTING AND SETTING ATTRIBUTES

```
class House
  attr_accessor :color
end

# instantiate a new object from the class
my_house = House.new

# set the color of the house with dot notation

my_house.color = 'Red'
my_house.color
=> 'Red'
```

# INITIALIZE

- This is a special method that determines the parameters for creating new instances with `.new`.
- The parameters passed into this method can set *instance variables*, which correspond to attributes.
- Initialize can also set defaults for new objects.

# INITIALIZE

```
class House
  attr_accessor :color, :stories, :rooms

  def initialize(stories, rooms, color)
    @stories = stories
    @rooms = rooms
    @color = color
  end
end

# instantiate a new object from the class
my_house = House.new(2, 3, 'blue'

my_house.color
=> 'blue'
```

# INITIALIZE

```
class House
  ............
  def to_s
     "This #{@color} house has #{@stories.to_s}
      stories and #{ @rooms.to_s } rooms."
  end
end

# instantiate a new object from the class
my_house = House.new(2, 3, 'blue')

puts my_house
"This blue house has 2 stories and 3 rooms."
```

# INITIALIZE

```
class House
  ............

  def paint(new_color)
    @color = new_color
  end
end

# instantiate a new object from the class
my_house.paint('red')

puts my_house
"This red house has 2 stories and 3 rooms."
```

# SEPARATE CLASSES, SEPARATE FILES

- You should require your files only as you need them from other files
- You can require these files with an internal ruby method require_relative “<relative path>”
- Unlike requiring gems ( remember `require 'httparty'`? ), you need to specify the relative path to the file.
- You don’t need `".rb"` at the end of the file path if you don’t want to.

# Inheriting traits and behavior

```ruby
# clones.rb

class Clone
  attr_accessor :hair_type, :accent, :needs_glasses, :special_skill

  def initialize (hair_type, accent, needs_glasses, special_skill)
    @hair_type = hair_type
    @accent = accent
    @needs_glasses = needs_glasses
    @special_skill = special_skill
    @is_awesome = true
  end

end

cosima = Clone.new('dreads', 'american', true, 'science')

puts cosima.inspect
#<Clone:0x007ff7221b7378 @hair_type="dreads", @accent="american", @needs_glasses=true, @is_awesome=true, @special_skill="science">
```

Think of a ball

# This could get out of hand

Imagine that you have a complex Ruby application that has a User class with lots of methods and attributes.

# This could get out of hand

If we want to make an Admin class that can do everything a user can do, but can also do some extra stuff, we don't have to rewrite everything from the User class.

# This could get out of hand

If we want to make an Admin class that can do everything a user can do, but can also do some extra stuff, we don't have to rewrite everything from the User class.

That would not be very elegant or easy to maintain.

# Rather than clones

What are some things that inherit traits from another?

For instance, a car and a bus share a lot in common before they diverge

```ruby

class Car
  attr_accessor :fuel_capacity, :max_speed, :gears,
     :num_passengers, :airbags, :engine_type, :
    manufacturer, :model, :owner_name

end


class Bus
  attr_accessor :fuel_capacity, :max_speed, :gears,
     :num_passengers, :airbags, :engine_type, :
    manufacturer, :model, :fare, :route, :
    transit_company

end
```

```ruby

class Car
  attr_accessor :fuel_capacity, :max_speed, :gears,
     :num_passengers, :airbags, :engine_type, :
    manufacturer, :model, :owner_name

end


class Bus
  attr_accessor :fuel_capacity, :max_speed, :gears,
     :num_passengers, :airbags, :engine_type, :
    manufacturer, :model, :fare, :route, :
    transit_company

end
```

only a few differences

```ruby
class Car
  attr_accessor :fuel_capacity, :max_speed, :gears,
     :num_passengers, :airbags, :engine_type, :
    manufacturer, :model, :owner_name

end


class Bus
  attr_accessor :fuel_capacity, :max_speed, :gears,
     :num_passengers, :airbags, :engine_type, :
    manufacturer, :model, :fare, :route, :
    transit_company

end
```

SO NOT DRY

only a few differences


INSTRUCTOR

Classes don’t just dictate behavior and traits for objects, they can pass them on to different classes

That’s how classes inherit from other classes

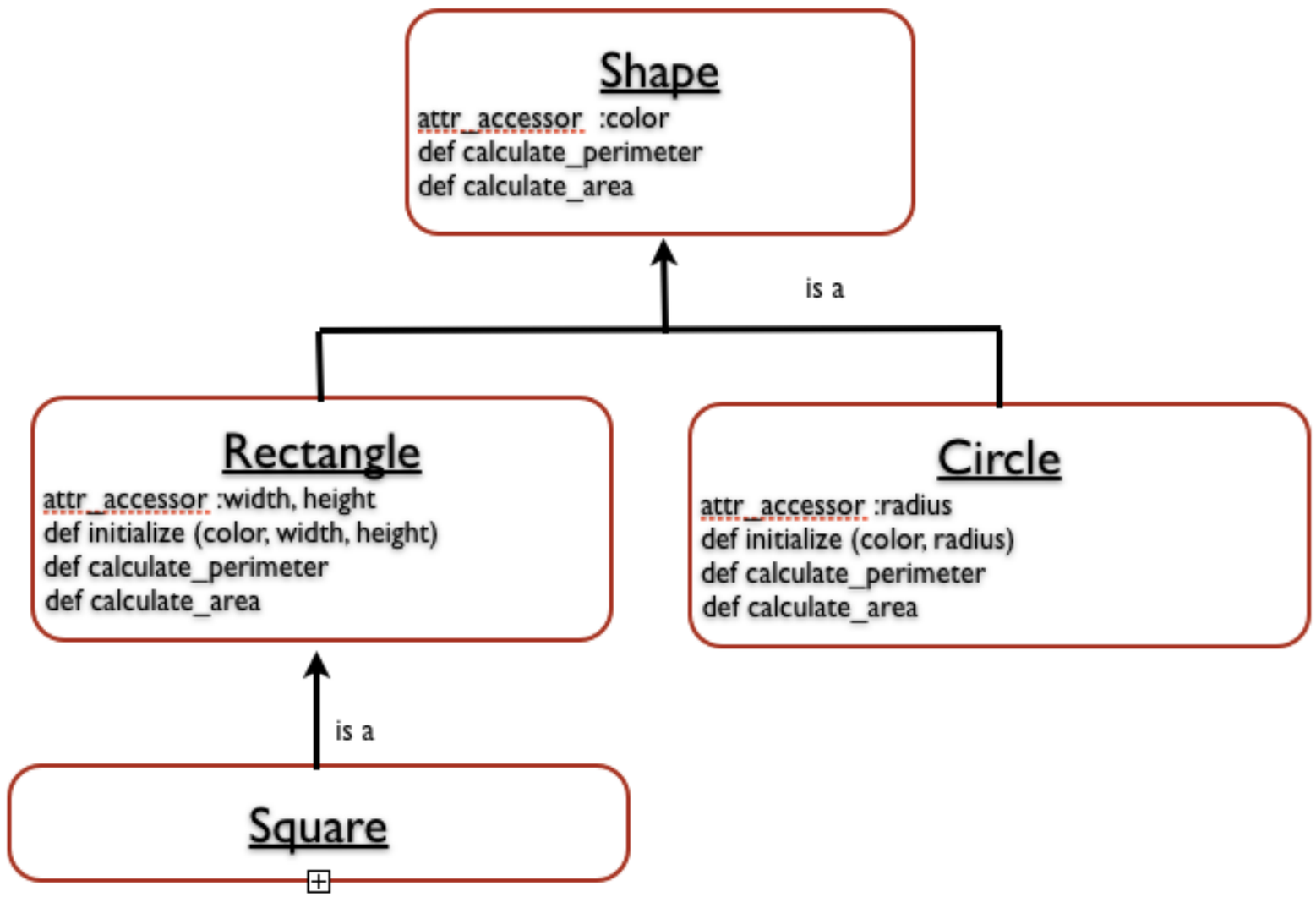
```
class MotorVehicle
  attr_accessor :fuel_capacity, :max_speed, :gears, :num_passengers, :airbags, :engine_type, :manufacturer, :model
end

class Car < MotorVehicle
  attr_accessor :owner_name

end

class Bus < MotorVehicle
  attr_accessor :fare, :route, :transit_company

end
```

To inherit from another class, use “<“

```ruby

class Car < MotorVehicle
  attr_accessor :owner_name

end


```

This could read like:

*I, Motor Vehicle, being of sound mind, do solemnly give thee, Car, all of my attributes and methods in perpetuity.*


Shape
attr_accessor :color
def calculate_perimeter
def calculate_area
is a
Rectangle
attr_accessor :width, height
def initialize (color, width, height)
def calculate_perimeter
def calculate_area
Circle
attr_accessor :radius
def initialize (color, radius)
def calculate_perimeter
def calculate_area
is a
Square

# How this is relevant to Rails

```ruby
class User < ActiveRecord::Base
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable
  has_many :memberships
  has_many :conversations, :through => :memberships
  has_many :messages, :through => :conversations
end
```

# Wrapping up

- What is a class?

# Wrapping up

- What is a class?
- What operator lets classes inherit from one another?

# Wrapping up

- What is a class?
- What operator lets classes inherit from one another?
- What are the benefits of inheritance?

# Wrapping up

- What is a class?
- What operator lets classes inherit from one another?
- What are the benefits of inheritance?
- How can a child class override its parent class?

- Classes can inherit from each other.
- When you inherit from another class, you gain <u>everything</u> about it. Minus the name.
  - This includes methods, attributes, class methods
- This is helpful for sharing functionality between classes, but sometimes you want two classes to share only some behavior.

# Modules

- Modules are what allow you to share only some behavior. You include them in classes. You can't make instances of modules
- They start with
  - module MyModuleName
- Similar to
  - class MyClassName

# Using them

- To compose a class with a module, you use the "include" method
- So, given a module called "Human"...
  - include Human
- "include" goes near your attr_accessor (if you have one)
- It should be inside the class

```ruby
module Think
  def ponder
    puts 'hmmm'
  end

  def draw_conclusions_from_empirical_observation
    puts "aha!"
  end
end

class Person
  include Think
end

class ArtificialIntelligence
  include Think
end

p = Person.new
a = ArtificialIntelligence.new

a.ponder
p.draw_conclusions_from_empirical_observation
```