METHOD INVOCATION IN RUBY

with
Maxwell
Mathews

WHAT WE'RE COVERING

- Basic class inheritance in Ruby
- Types of Ruby methods
  - Class
  - Instance
  - Singleton
  - Module
- Method invocation based on inheritance rules

MY APPROACH

- Start with a high level and work downwards
  - I like to examine some big picture examples, break them down into their constituent parts, and put them all back together
  - It's how I approach my professional troubleshooting and personal projects
- Work with and break down some simple classes (hopefully not too contrived) and explore their relationships
- What I hope you'll get out of this discussion
  - An understanding of what Ruby is doing when your code makes method calls under various circumstances
  - The ability to trace through your code when method naming conflicts arise

WHO I AM

- Software Engineer with 8 years of object-oriented design and implementation experience
- Systems Integration Engineer specializing in automating the runtime behavior of distributed simulation systems
- Trainer and curriculum designer for support engineers who maintain my company's deployed trainers
- A lover of complex (and complicated) systems

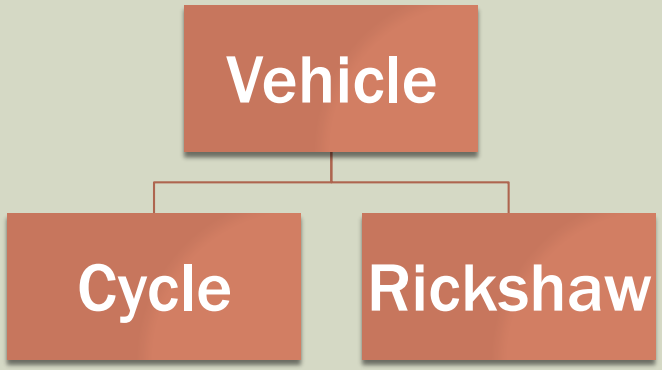
LET'S START!

WHAT IS THIS ALL ABOUT?

maxBike singleton → Shiftable Module proxy class → Portable Module proxy class → Cycle class → Rideable Module proxy class → Rollable Module proxy class → Vehicle class

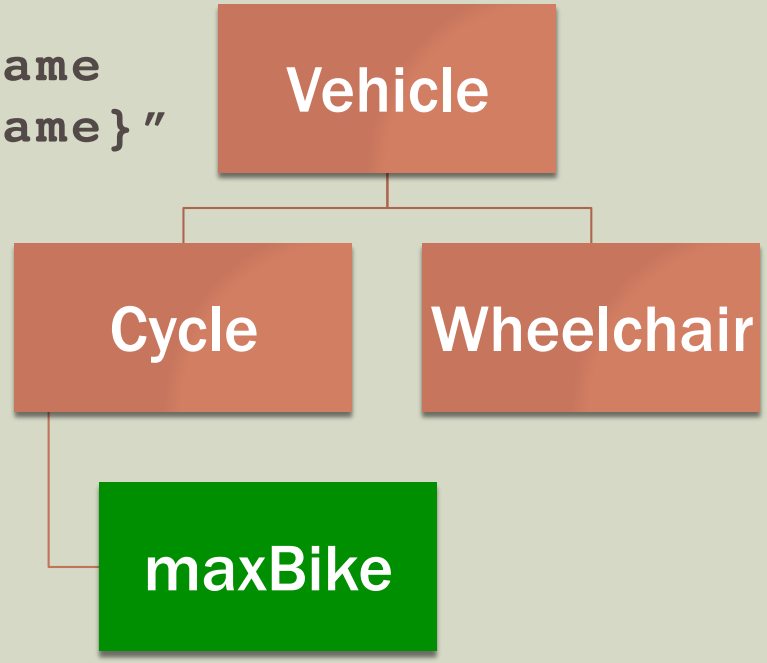
RUBY CLASSES

```
class Vehicle
    def self.state_my_class_name
        return "I'm a #{self.name}"
    end
    def roll
        return "I'm rolling"
    end
end

class Cycle < Vehicle
    ...
end

class Rickshaw < Vehicle
    ...
end
```



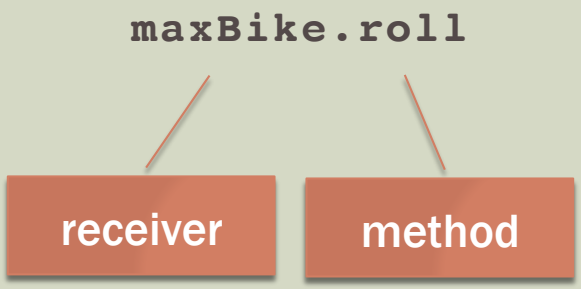
RUBY OBJECT INSTANCES

```
class Vehicle
    def self.state_my_class_name
        return "I'm a #{self.name}"
    end
    def roll
        return "I'm rolling"
    end
end

class Cycle < Vehicle
    ...
end

maxBike = Cycle.new
```



OBJECT CALLS

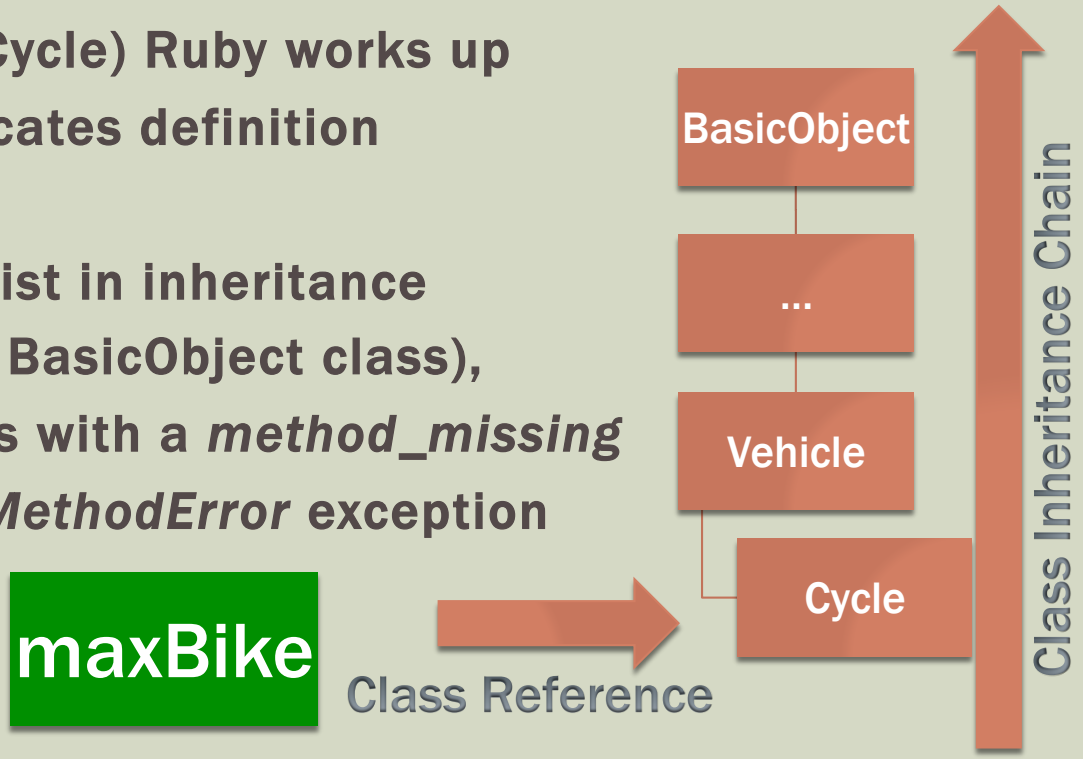
- Per Ruby Documentation
  - In an object call, there is a **method** and a **receiver**



  - **Methods are contained in classes, not in individual objects (instances of a class)**
  - **Individual objects consist of three main parts**
    - **Instance variables**
    - **Object flags**
    - **Associated class – CRITICAL TO METHOD RESOLUTION**
  - **N.B. Ruby does not allow for method overloading, but only allows a single method of a given name per class with optional arguments**

OBJECT CALLS

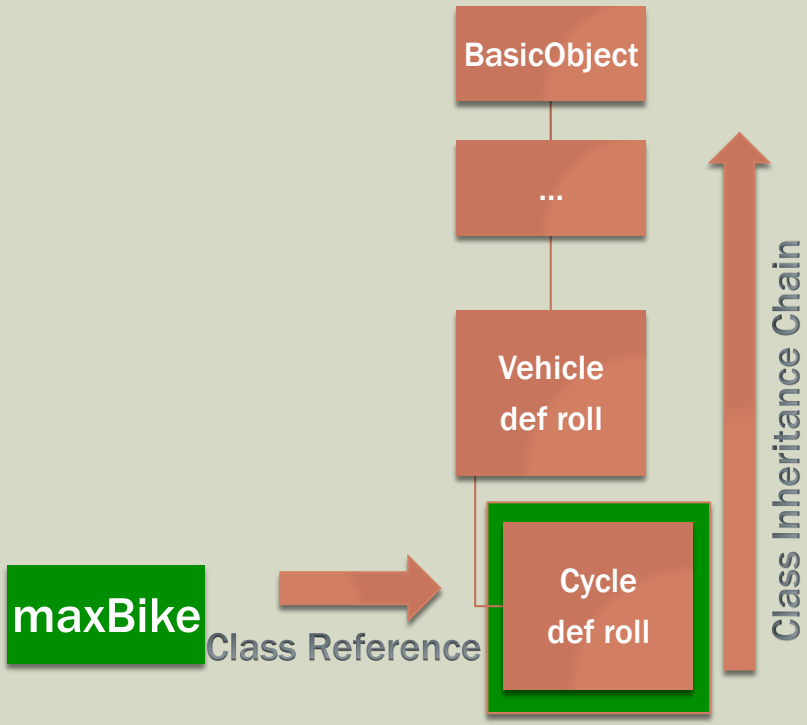
- How does `maxBike.roll` get resolved?
- **The method does not exist in the "Cycle" class!**
- **Ruby interpreter determines class of** maxBike object
- **After determining class (Cycle) Ruby works up inheritance chain until it locates definition of** "roll"
- **If the method does not exist in inheritance chain (all the way up to the BasicObject class), the receiver object responds with a *method_missing* message, resulting in a *NoMethodError* exception**



METHOD OVERRIDING

- What if we want the "roll" method to do something specific for "Cycle" objects?
- Override the method

```
class Vehicle
    def self.state_my_class_name
        return "I'm a #{self.name}"
    end
    def roll
        return "I'm rolling"
    end
end

class Cycle < Vehicle
    def roll
        return "I'm a rolling cycle"
    end
end
```



- Calling `maxBike.roll` still looks at inheritance chain to locate method name, but Ruby interpreter finds the name lower in the chain (in "Cycle" class) and executes that code block instead

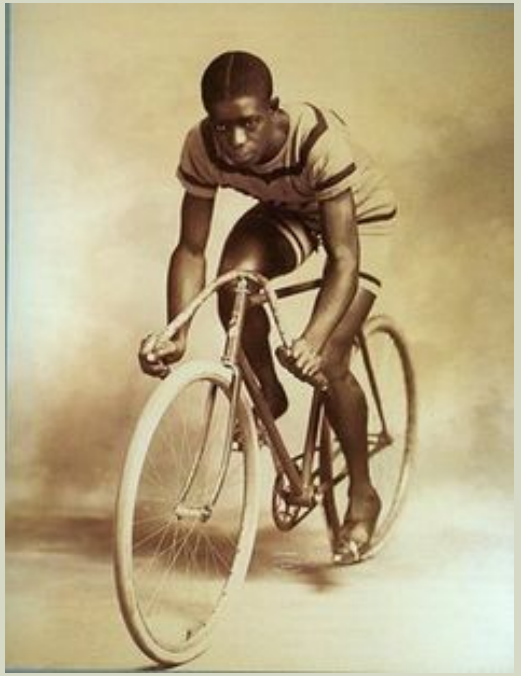
METHOD OVERRIDING

- Methods are functions contained within classes and are associated with individual objects (preview: how does this work when everything in Ruby is an object?)
- When an object instance (receiver) is sent a message (method), Ruby searches up an object's class hierarchy to find the first definition of the method and runs that code
- Even if the method is defined at multiple points in the class hierarchy, the first definition is always run exclusively.
- Use the keyword "super" within an overridden method to call the next highest equivalent method in the inheritance chain
  - Also good for debugging inheritance hierarchy

SINGLETON METHODS

- What if we want to define an object-specific method?
  - maxBike is SO special, it has its own “roll” method

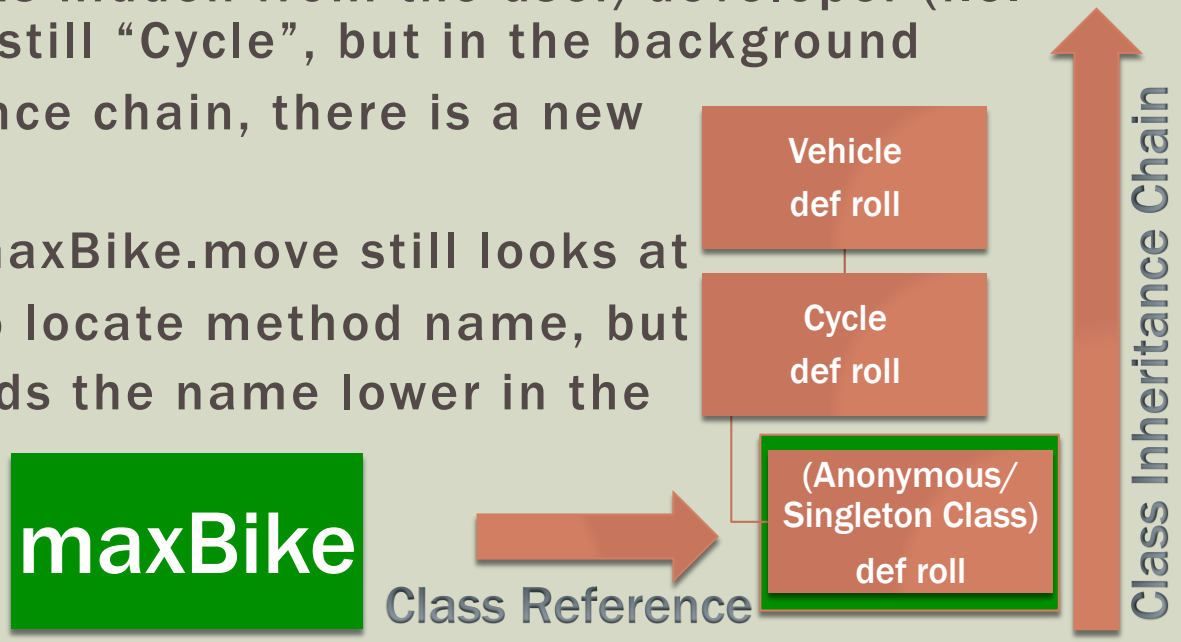
```
maxBike = Cycle.new

class << maxBike
   def roll
      “Max’s bike is rolling”
   end
end

# OR

def maxBike.roll
      “Max’s bike is rolling”
end
```



- But I thought methods were only defined in classes not objects?!
- What happens to our inheritance hierarchy?

SINGLETON METHODS

- Methods *ARE* only defined in classes!
- Enter the magical singleton class (or anonymous class)
- To maintain method definitions inside class definitions, Ruby creates a *singleton* class specific to the maxBike instance
- The singleton class is hidden from the user/developer (i.e. `maxBike.class` is still "Cycle", but in the background Ruby class inheritance chain, there is a new class)
- As before, Calling maxBike.move still looks at inheritance chain to locate method name, but Ruby interpreter finds the name lower in the chain (in `maxBike` singleton class) and executes that code block instead



SINGLETON METHODS

- Singleton methods (and their associated classes) are a mechanism to define unique behavior for individual objects

CLASS METHOD RESOLUTION

- Everything in Ruby is an object
  - All datatypes, *including classes*, are objects
  - Confusingly, class definitions are instances of the class "Class"
- All method execution is performed in the context of the current "self", which is the current object
  - `maxBike.move # -> "self" is now maxBike`
  - If you're in a class definition

```
class Vehicle
   def self.state_my_class_name
      return "I'm a #{self.name}"
   end
end

#or

class Vehicle
   class << self
       def class_type
          return "I'm a #{self.name}"
       end
   end
end
```

"self" is the "Class" object named "Vehicle"

CLASS METHOD RESOLUTION

```
class Vehicle
   def self.state_my_class_name
      return "I'm a #{self.name}"
   end
end

#or

class Vehicle
   class << self
       def state_my_class_name
          return "I'm a #{self.name}"
       end
   end
end
```

- Discussion: What is happening when we define "Vehicle" and its "state_my_class_name" class method?
- Hints:
  - What does "self" represent in the context of a class definition?
  - Understanding what "self" represents, what of what we've seen does this construction (`def self.state_my_class_name`) resemble?

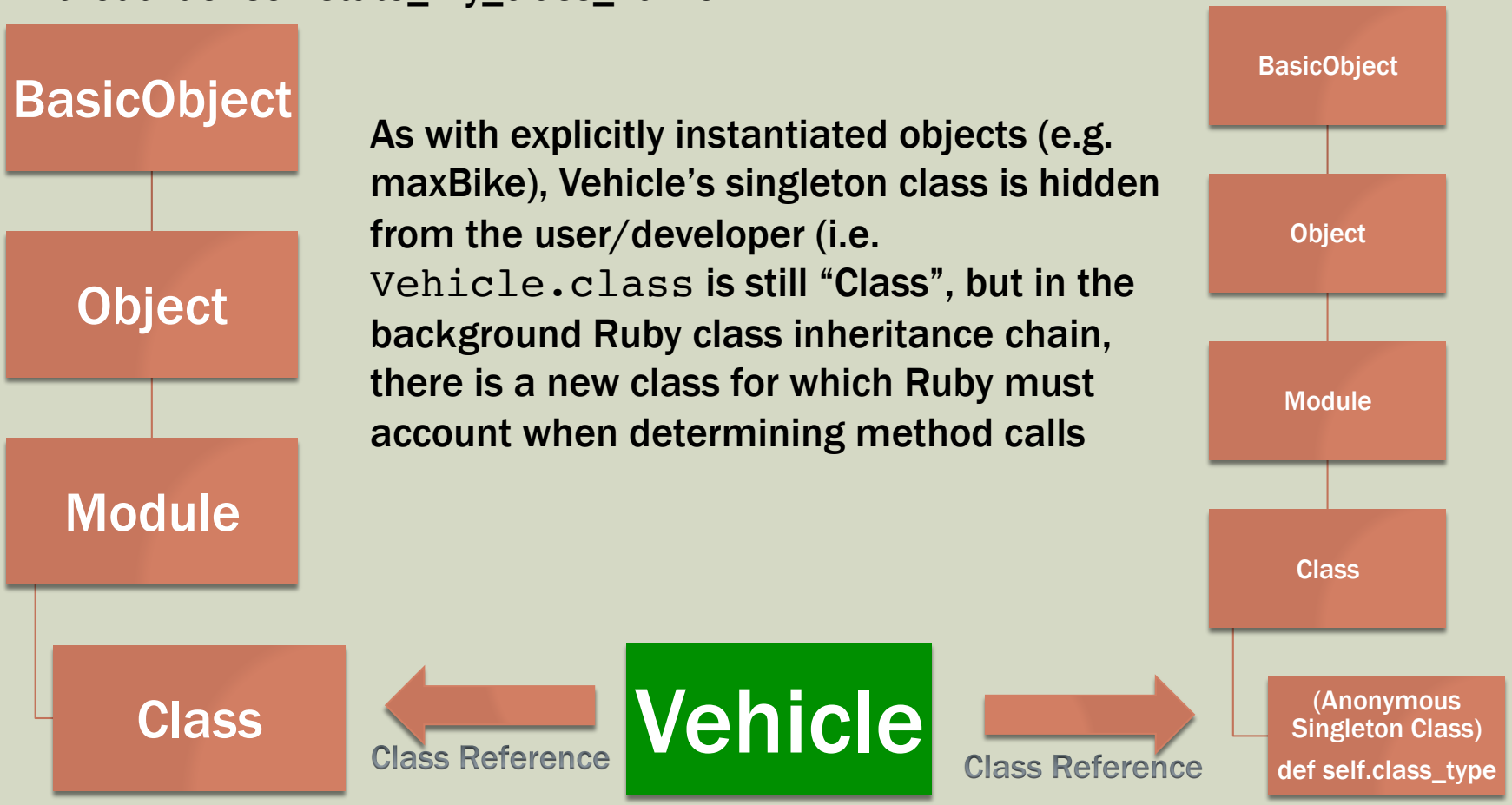
CLASS METHOD RESOLUTION

- "self" in the context of our class definition represents the current object: the instance of the "Class" object called "Vehicle"
- self.state_my_class_name == Vehicle.state_my_class_name
  - N.B. it is syntactically valid and semantically equivalent to define our class method as "Vehicle.state_my_class_name" inside our class defintion
- Referencing "Vehicle" causes Ruby to look up the "Vehicle" "Class" object and its associated class hierarchy
- Recall: Individual objects consist of three main parts
  - Instance variables
  - Object flags
  - *Associated class*
- Defining a method on "self" (the "Vehicle" object) causes Ruby to create a singleton class that is inserted into the inheritance chain
- Thought/coding experiment: define a superclass/class relationship each with identical class methods (e.g. self.do_something). What happens when you call "super" inside the child class? Why is that behavior occurring?

CLASS METHOD RESOLUTION

Without “def self.state_my_class_name”

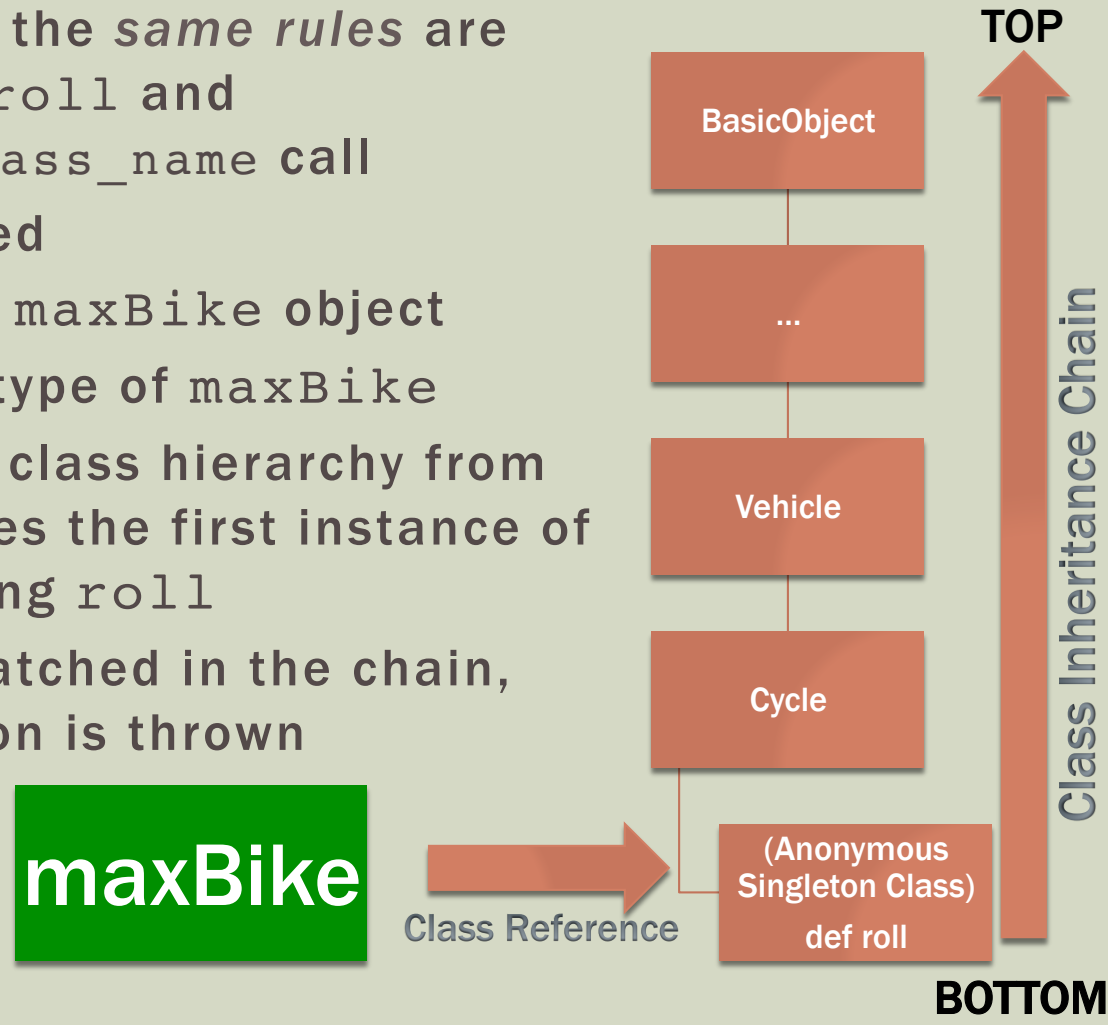
With “def self.state_my_class_name”



As with explicitly instantiated objects (e.g. maxBike), Vehicle’s singleton class is hidden from the user/developer (i.e. `Vehicle.class` is still “Class”, but in the background Ruby class inheritance chain, there is a new class for which Ruby must account when determining method calls

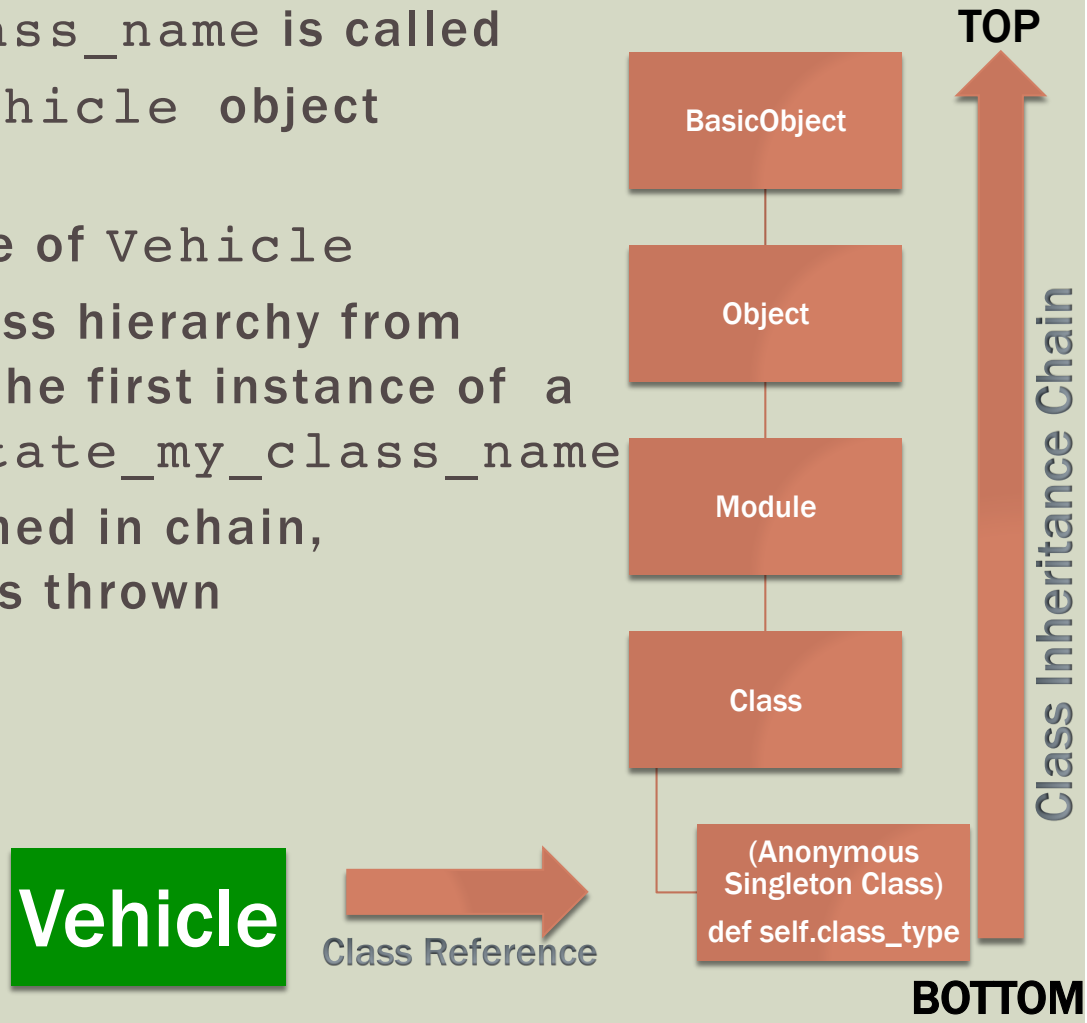
WHAT'S HAPPENING?

- In the Ruby interpreter, the *same rules* are applied to a `maxBike.roll` and `Vehicle.state_my_class_name` call
- `maxBike.roll` is called
- `self` is changed to the `maxBike` object
- Ruby locates the class type of `maxBike`
- Ruby steps through the class hierarchy from bottom up until it locates the first instance of a method name matching `roll`
- If a method can't be matched in the chain, *NoMethodError* exception is thrown



WHAT'S HAPPENING?

- `Vehicle.state_my_class_name` is called
- `self` is changed to the `Vehicle` object (Instance of "Class" class)
- Ruby locates the class type of `Vehicle`
- Ruby steps through the class hierarchy from bottom up until it locates the first instance of a method name matching `state_my_class_name`
- If a method can't be matched in chain, *NoMethodError* exception is thrown



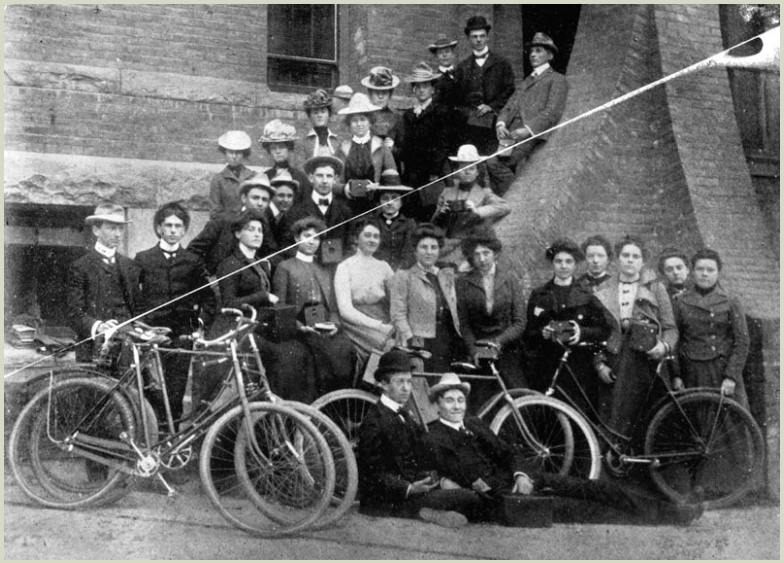
CLASS METHOD RESOLUTION

- Ruby moves “out” from an object into its class and work “up” the inheritance chain to resolve method names

INCLUDING MODULES

- Instead of defining a “roll” method in our class hierarchy, we’ll be a little more astute in our architecture
- We’ll define two modules, “Rollable” and “Rideable” which will be used by our classes
- Since more than just Vehicles “roll”
  - It makes sense to encapsulate our methods into a Module
  - Also creating a class such as “RollingThing” doesn’t make sense

```
module Rollable
   def roll
      return “I’m rolling”
   end
end

module Rideable
   def ride
      return “I’m rideable”
   end
end
```



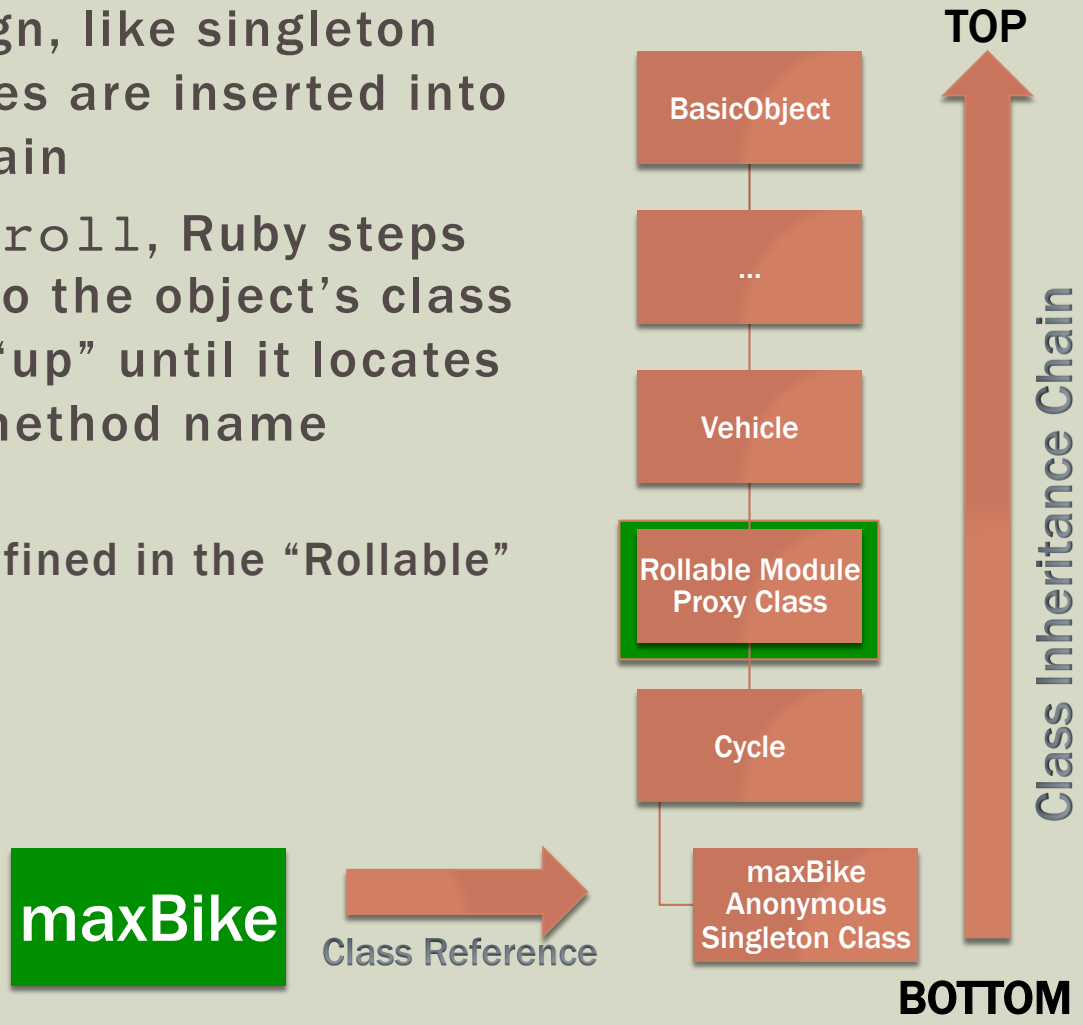
INCLUDING MODULES

- Including modules into a class definition turns the method definitions inside the module into instance methods for instances of the including class, i.e.

```
module Rollable
   def roll
      return "I'm rolling"
   end
end

class Vehicle
end

class Cycle < Vehicle
   include Rollable
end

maxBike = Cycle.new

class << maxBike
   def go_fast
      "Max's bike is flying!"
   end
end

puts maxBike.roll #=> prints "I'm rolling"
puts maxBike.go_fast #=> prints "Max's bike is flying!"
```

INCLUDING MODULES

- To satisfy the Ruby design, like singleton classes, included modules are inserted into the class inheritance chain
- When calling `maxBike.roll`, Ruby steps "out" from the object into the object's class hierarchy and searches "up" until it locates the first instance of a method name matching `roll`
  - In our new code, roll is defined in the "Rollable" module



EXTENDING MODULES

- What happens when you extend modules on class
  - Module methods are added as class methods

```
module Rollable
   def roll
      return "I'm rolling"
   end
end


class Cycle < Vehicle
   extend Rollable
end

puts Cycle.roll #=> prints "I'm rolling"
```

- Discussion:
  - What happens when class methods are created?
  - What does the "Class" object's inheritance chain look like after extending modules from within a class definition

- Discussion: what happens when you extend modules on singleton class, e.g. `maxBike.extend(Shiftable)`

METHOD RESOLUTION RULES

- Review
  - All method execution is performed in the context of the current “self”, which is the current object
  - After setting “self” the program moves “out” from the object and “up” the object’s class hierarchy
  - Once Ruby determines and sets up the inheritance chain, method resolution follows from bottom up

METHOD RESOLUTION RULES

What rules define the class hierarchy?

METHOD RESOLUTION RULES

- The highest priority class is the singleton class, which is placed at the bottom of the inheritance chain

METHOD RESOLUTION RULES

- Next are modules mixed into the singleton class in reverse order of inclusion (extended)

```
maxBike.extend(Portable)
maxBike.extend(Shiftable)
```

yields

METHOD RESOLUTION RULES

- Next comes the object's class (for `maxBike`, the `Cycle` class) and any instance methods defined in the class

METHOD RESOLUTION RULES

- Next are modules included into the object's class in reverse order of inclusion

```
class Cycle
    include (Rollable)
    include (Rideable)
end
```

yields

METHOD RESOLUTION RULES

- The object's superclass.

- Obviously, this process is recursive
- If Vehicle has included modules or super classes, those will be incorporated into the chain as well

METHOD RESOLUTION RULES

- All method execution is performed in the context of the current "self", which is the current object
- After setting "self" the program moves "out" from the object and "up" the object's class hierarchy
- Once Ruby determines and sets up the inheritance chain, method resolution follows from bottom up
- Experiment with "super" call to see relationships for yourself
- Methods are resolved in the following order based on the inheritance chain rules (taken from reference [1])
  - Methods defined in the object's singleton class (i.e. the object itself)
  - Modules mixed into the singleton class in reverse order of inclusion
  - Methods defined by the object's class
  - Modules included into the object's class in reverse order of inclusion
  - Methods defined by the object's superclass
    - N.B. The key is that if there are multiple identical method definitions in an inheritance tree, the above rules are used to determine the one that will be run

FURTHER READING/REFERENCES

- Gregory Brown, “Ruby’s method lookup path, Part 1”, https://www.practicingruby.com/articles/method-lookup-1 [1]
- Gregory Brown, “Ruby’s method lookup path, Part 2”, https://www.practicingruby.com/articles/method-lookup-2 [2]
- Peter J. Jones, “Understanding Ruby Singleton Classes”, http://www.devalot.com/articles/2008/09/ruby-singleton [3]
- “Programming Ruby: The Pragmatic Programmer’s Guide (Classes and Objects), http://ruby-doc.com/docs/ProgrammingRuby/ [4]