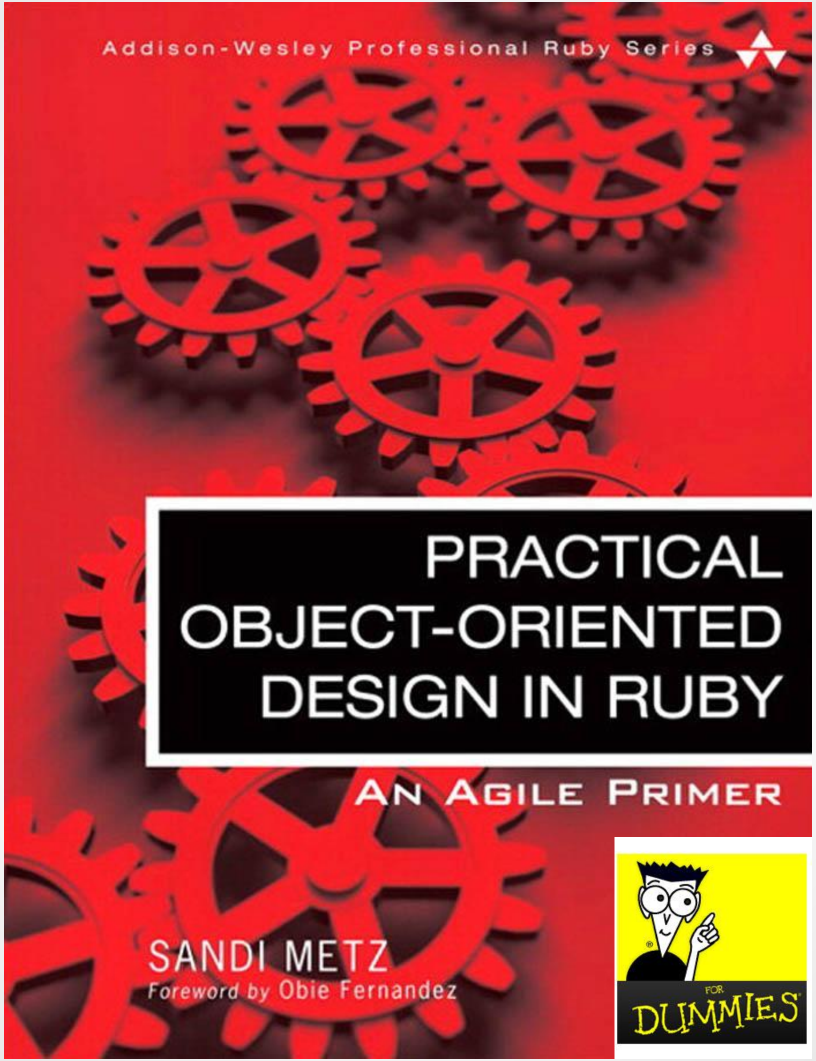

Addison-Wesley Professional Ruby Series
PRACTICAL OBJECT-ORIENTED DESIGN IN RUBY
AN AGILE PRIMER
SANDI METZ
Foreword by Obie Fernandez
FOR DUMMIES

The principal problem faced in design:

Writing Code for Today’s Feature AND Tomorrow’s Feature

Easy To Change Code Is

- **Transparent**: Consequences of change should be obvious in the code, and in distant code that relies upon it.
- **Reasonable**: Marginal Cost of Change <= Marginal Benefits of Change
- **Usable**: Should be usable in new and unexpected contexts
- **Exemplary**: Encourages those who change it to perpetuate these qualities

Object Oriented Code Isn’t About Objects.

Object Oriented Code is about Messages

OVERVIEW

Single Responsibility Classes

Dependencies

Flexible Interfaces

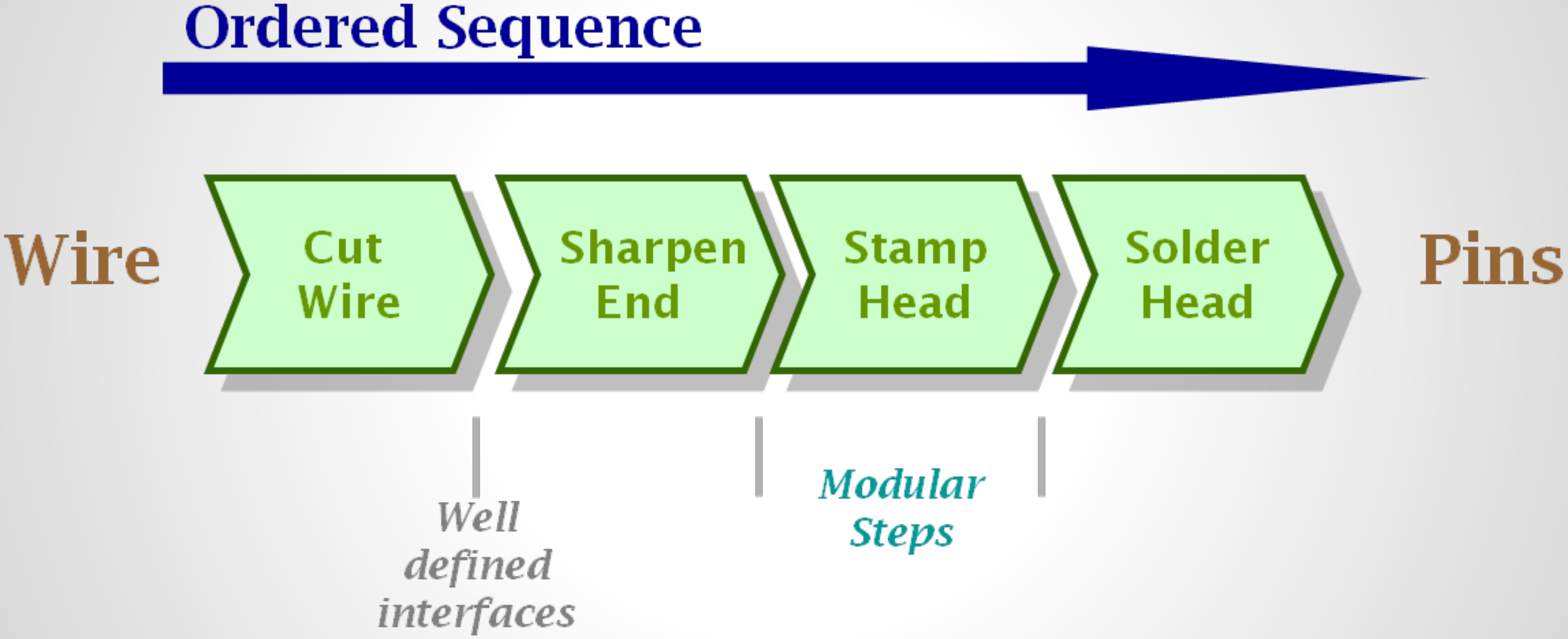
Duck Typing

Inheritance

Modules

Combining Objects with Composition

Single Responsibility Principle



Ordered Sequence

Wire

Cut Wire

Sharpen End

Stamp Head

Solder Head

Pins

Well defined interfaces

Modular Steps

Single Responsibility

Why does it matter?

- Classes with more responsibilities have more opportunities to break!
- More entangling of responsibilities within the class
- Harder to access only code you need when you reuse, and requires duplication of code.

Single Responsibility

How do we know it’s got more than one responsibility?

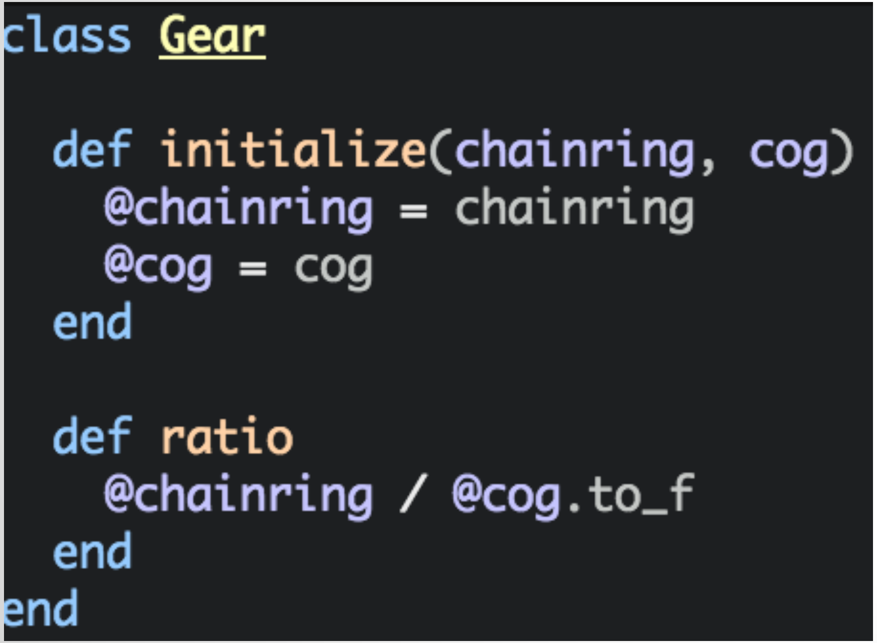
- Ask it, in plain language, its responsibilities
  - “Mr. Gear, what is your tire(size)?”
- Describe it, in plain language
  - If you have to use conjunctions, you probably don’t have a single responsibility

Single Responsibility

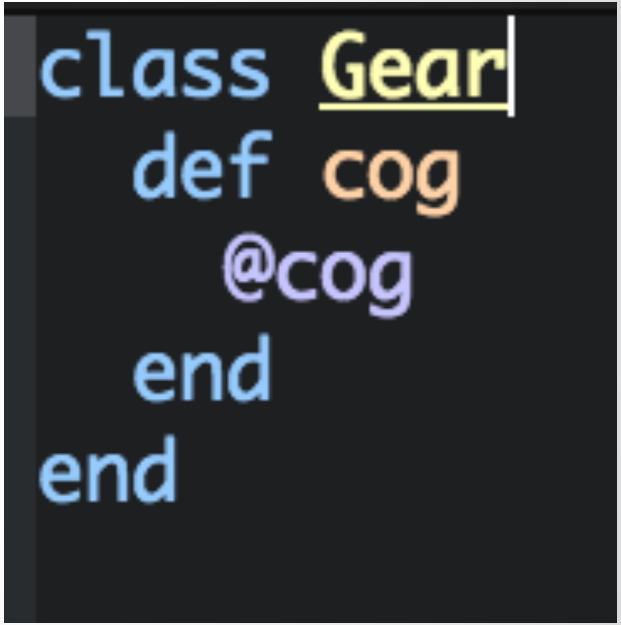
How do we Write code that embraces change?

- Depend on Behavior, Not on Data
  - Hide Instance Variables- Make them Behavior
  - Hide Data Structures
- Enforce Single Responsibility Everywhere

Hiding Instance Variables

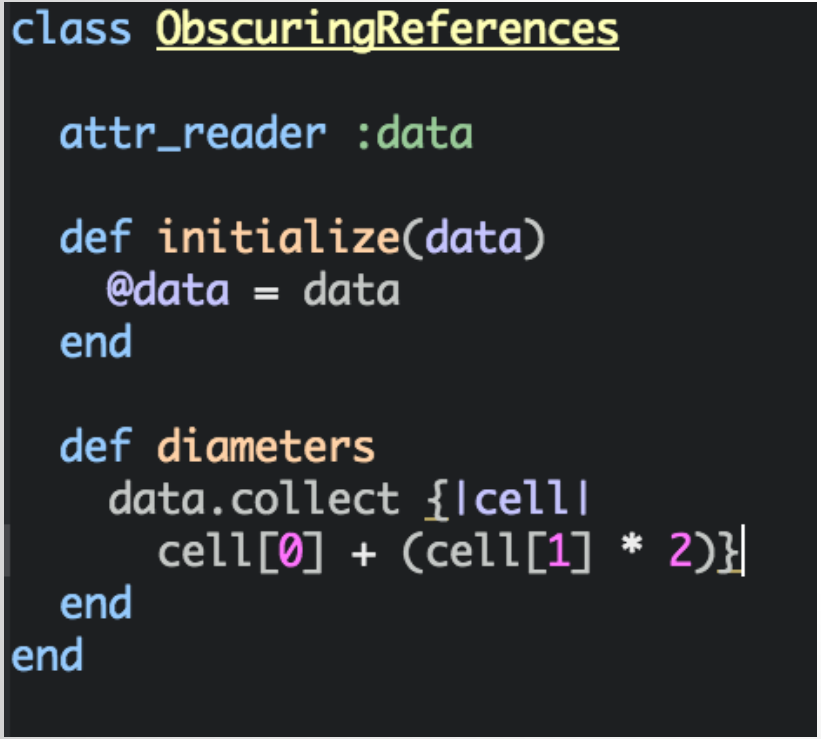


```ruby
class Gear

  def initialize(chainring, cog)
    @chainring = chainring
    @cog = cog
  end

  def ratio
    @chainring / @cog.to_f
  end
end
```

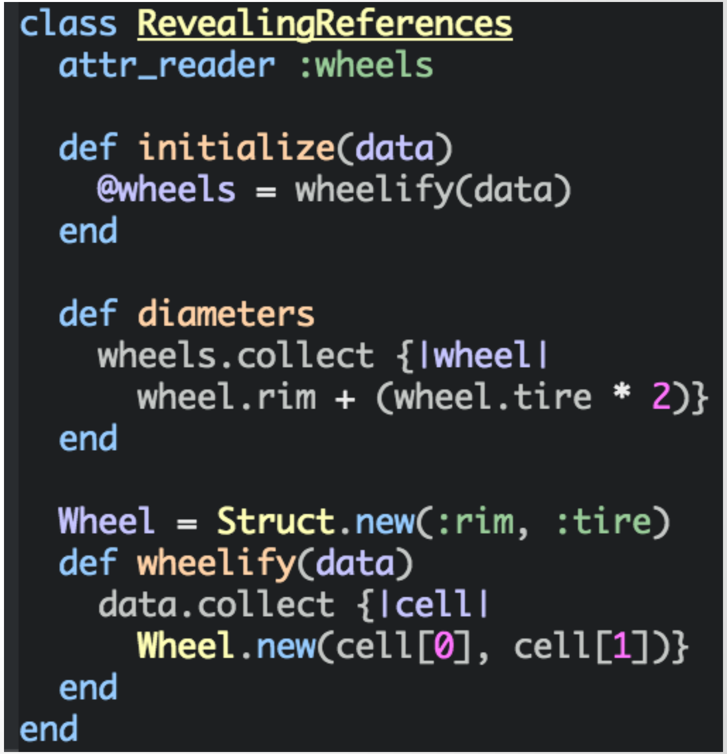


```ruby
class Gear
  def cog
    @cog
  end
end
```

Hiding Data Structures

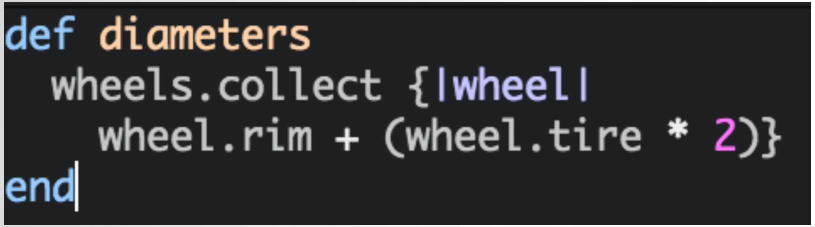


```ruby
class ObscuringReferences

  attr_reader :data

  def initialize(data)
    @data = data
  end

  def diameters
    data.collect {|cell|
      cell[0] + (cell[1] * 2)}
  end
end
```

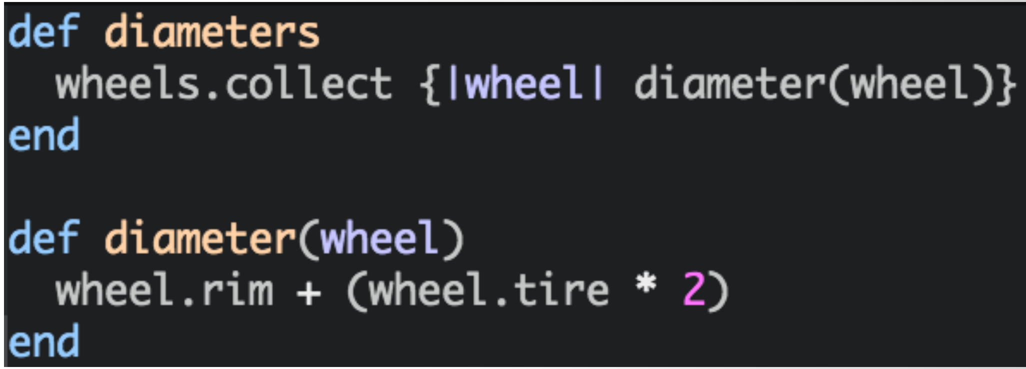


```ruby
class RevealingReferences
  attr_reader :wheels

  def initialize(data)
    @wheels = wheelify(data)
  end

  def diameters
    wheels.collect {|wheel|
      wheel.rim + (wheel.tire * 2)}
  end

  Wheel = Struct.new(:rim, :tire)
  def wheelify(data)
    data.collect {|cell|
      Wheel.new(cell[0], cell[1])}
  end
end
```

Enforcing Single Responsibility Everywhere



```ruby
def diameters
  wheels.collect {|wheel|
    wheel.rim + (wheel.tire * 2)}
end
```



```ruby
def diameters
  wheels.collect {|wheel| diameter(wheel)}
end

def diameter(wheel)
  wheel.rim + (wheel.tire * 2)
end
```

Exposes Previously Hidden Qualities
Avoids Comments
Encourages Reuse
Easy to Move to Other Classes

Managing Dependencies

SO IF I NEED TO STOP SMOKING AND LOSE WEIGHT.. WHAT ARE YOU GOING TO DO ABOUT IT?
FRAN

Managing Dependencies

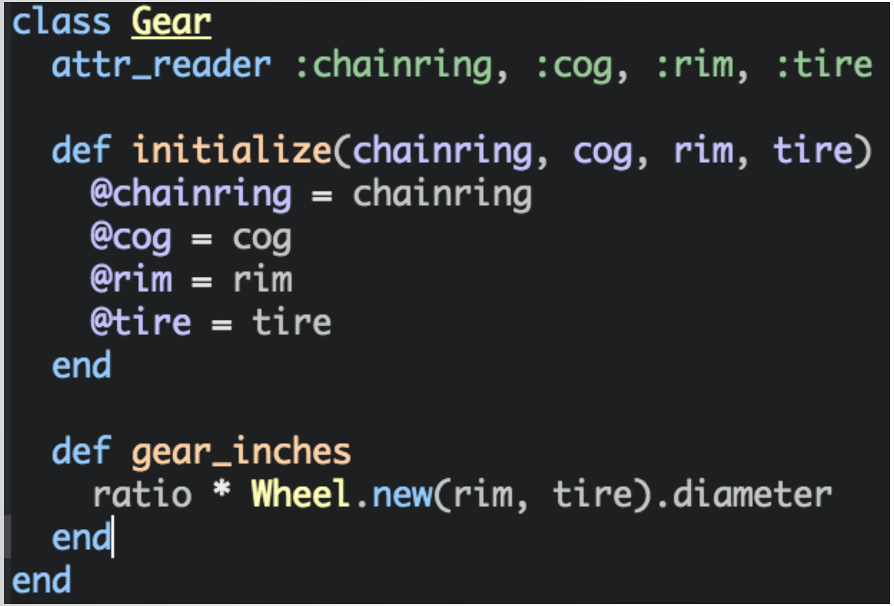
Dependencies Occur when an object knows:

- The name of another class
- Name of message it will send to something other than `self`
- Arguments a message requires
- The order of those arguments

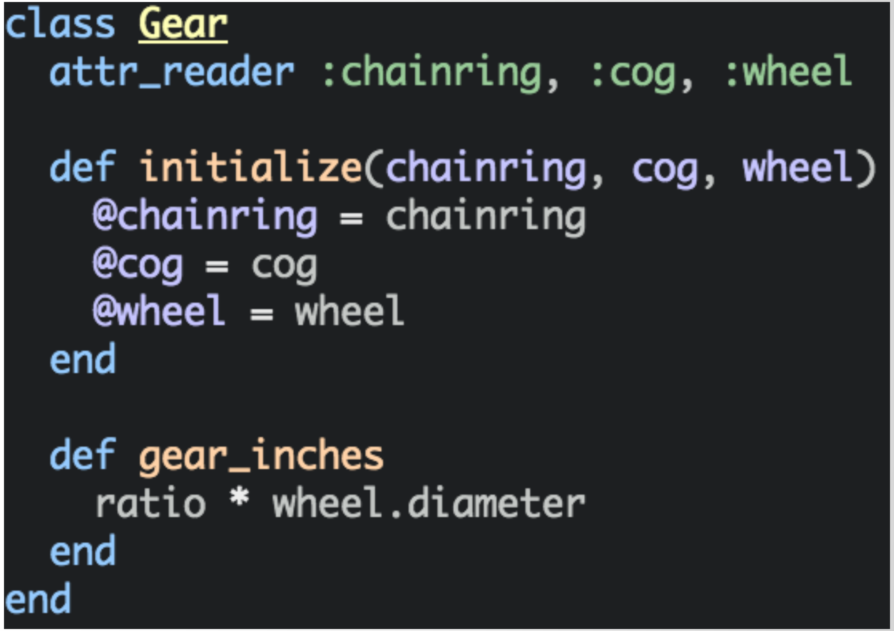
Managing Dependencies

- Inject Dependencies (Duck Typing)
- Isolate Dependencies
  - Instance Creation
  - Vulnerable External Messages
- Argument-Order Dependencies
  - Using Hashes for Arguments

Inject Dependencies

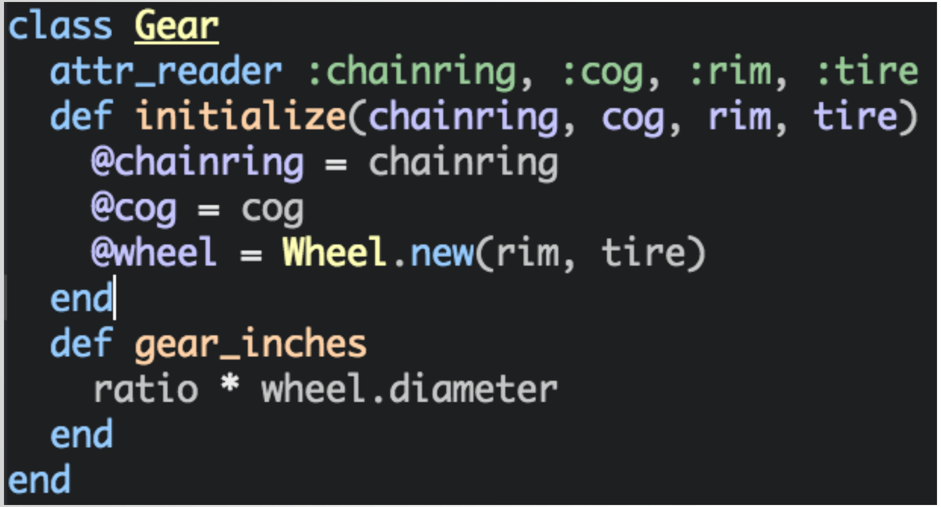


```ruby
class Gear
  attr_reader :chainring, :cog, :rim, :tire

  def initialize(chainring, cog, rim, tire)
    @chainring = chainring
    @cog = cog
    @rim = rim
    @tire = tire
  end

  def gear_inches
    ratio * Wheel.new(rim, tire).diameter
  end
end
```

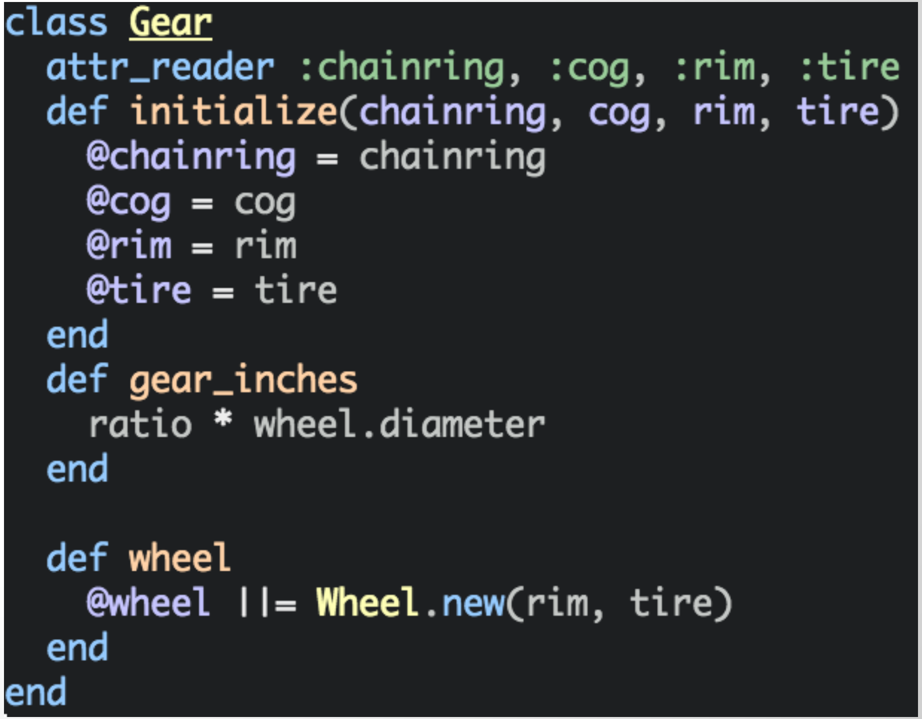


```ruby
class Gear
  attr_reader :chainring, :cog, :wheel

  def initialize(chainring, cog, wheel)
    @chainring = chainring
    @cog = cog
    @wheel = wheel
  end

  def gear_inches
    ratio * wheel.diameter
  end
end
```

Isolate Instance Creation

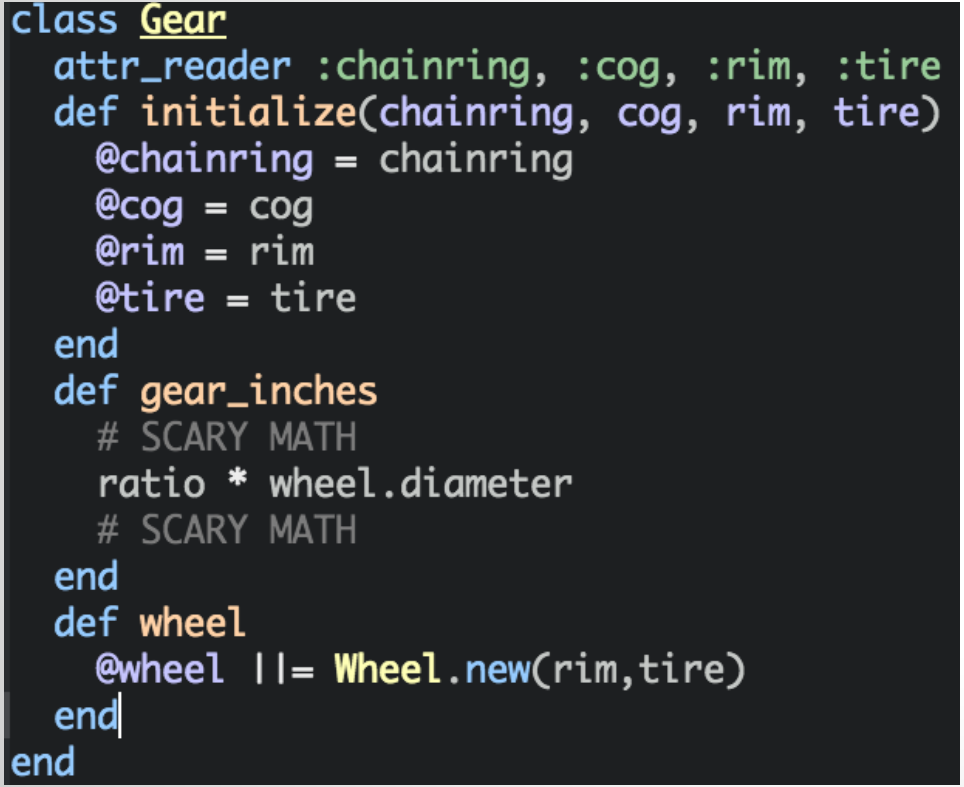


```ruby
class Gear
  attr_reader :chainring, :cog, :rim, :tire
  def initialize(chainring, cog, rim, tire)
    @chainring = chainring
    @cog = cog
    @wheel = Wheel.new(rim, tire)
  end
  def gear_inches
    ratio * wheel.diameter
  end
end
```

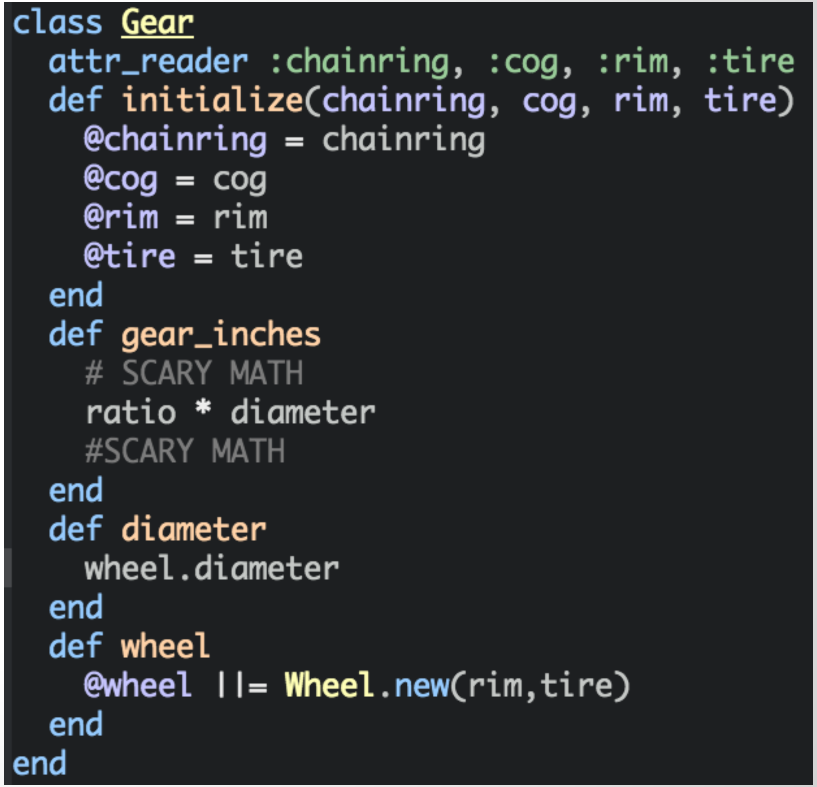


```ruby
class Gear
  attr_reader :chainring, :cog, :rim, :tire
  def initialize(chainring, cog, rim, tire)
    @chainring = chainring
    @cog = cog
    @rim = rim
    @tire = tire
  end
  def gear_inches
    ratio * wheel.diameter
  end

  def wheel
    @wheel ||= Wheel.new(rim, tire)
  end
end
```

Isolate Vulnerable External Messages

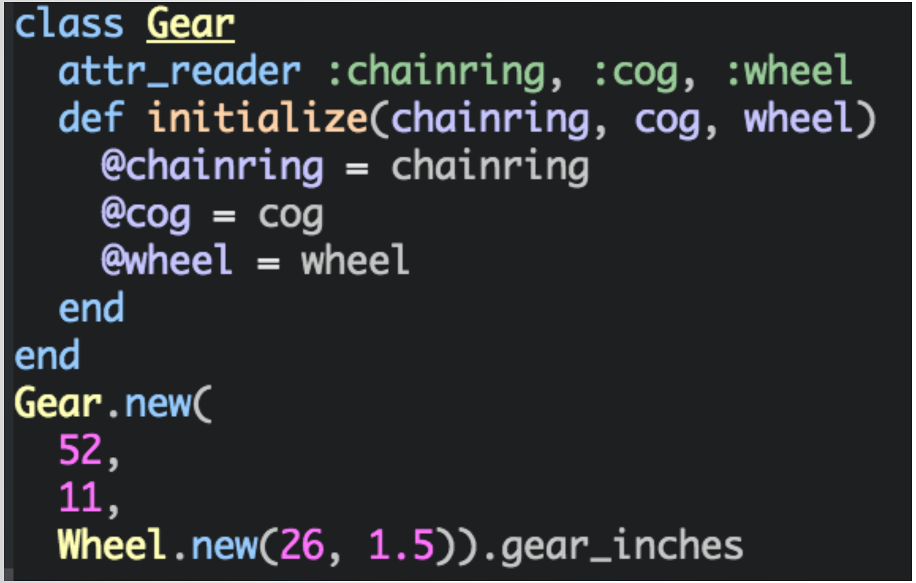


```ruby
class Gear
  attr_reader :chainring, :cog, :rim, :tire
  def initialize(chainring, cog, rim, tire)
    @chainring = chainring
    @cog = cog
    @rim = rim
    @tire = tire
  end
  def gear_inches
    # SCARY MATH
    ratio * wheel.diameter
    # SCARY MATH
  end
  def wheel
    @wheel ||= Wheel.new(rim,tire)
  end
end
```

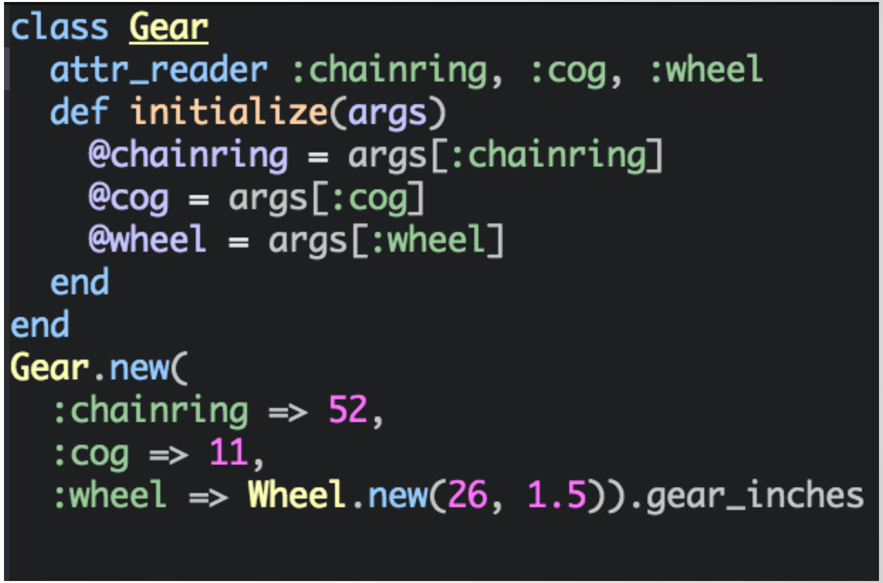


```ruby
class Gear
  attr_reader :chainring, :cog, :rim, :tire
  def initialize(chainring, cog, rim, tire)
    @chainring = chainring
    @cog = cog
    @rim = rim
    @tire = tire
  end
  def gear_inches
    # SCARY MATH
    ratio * diameter
    #SCARY MATH
  end
  def diameter
    wheel.diameter
  end
  def wheel
    @wheel ||= Wheel.new(rim,tire)
  end
end
```

Remove Argument-Order Dependencies



```ruby
class Gear
  attr_reader :chainring, :cog, :wheel
  def initialize(chainring, cog, wheel)
    @chainring = chainring
    @cog = cog
    @wheel = wheel
  end
end
Gear.new(
  52,
  11,
  Wheel.new(26, 1.5)).gear_inches
```



```ruby
class Gear
  attr_reader :chainring, :cog, :wheel
  def initialize(args)
    @chainring = args[:chainring]
    @cog = args[:cog]
    @wheel = args[:wheel]
  end
end
Gear.new(
  :chainring => 52,
  :cog => 11,
  :wheel => Wheel.new(26, 1.5)).gear_inches
```

Creating Flexible Interfaces

SCOTT HALLERAN/GETTY IMAGES

Creating Flexible Interfaces

Private Interface

- Handle implementation details
- Are not expected to be sent by other objects
- Can change for any reason whatsoever
- May not even be referenced in tests

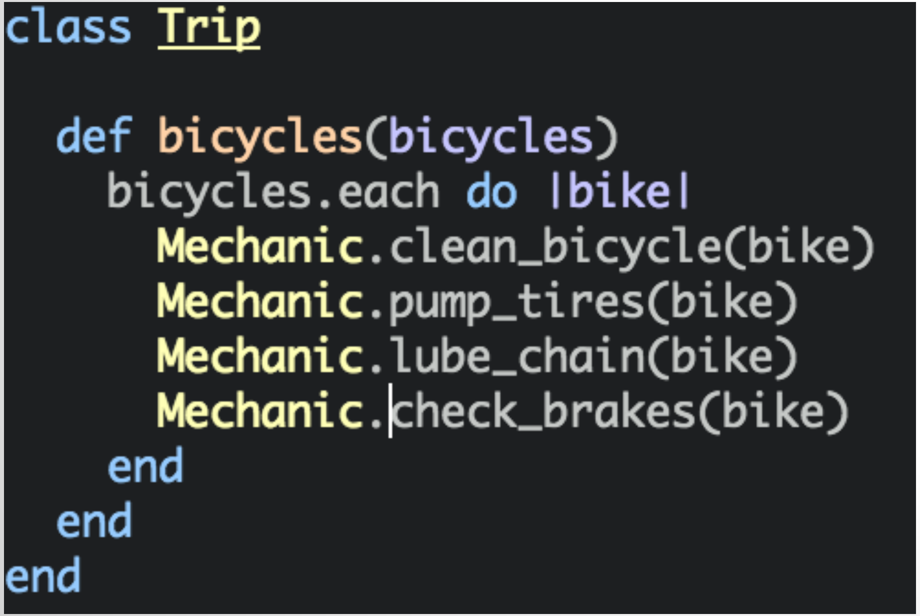
Creating Flexible Interfaces

Public Interface

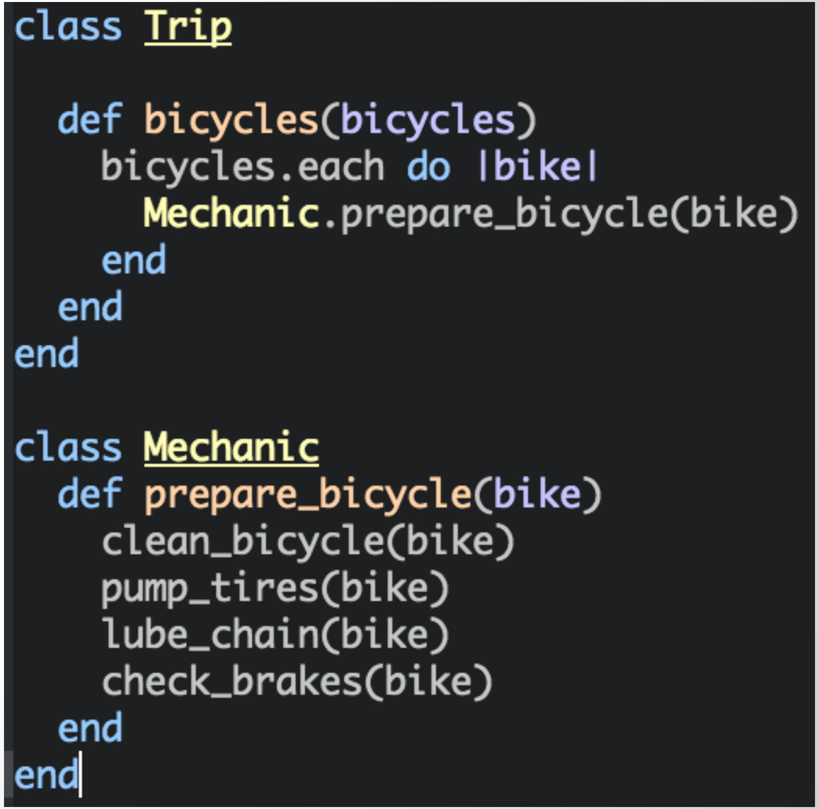
- Reveal primary responsibility
- Expect to be invoked by others
- Don’t change on a whim
- Safe for others to depend on
- Thoroughly documented in tests

Creating Flexible Interfaces

Tell instead of Ask

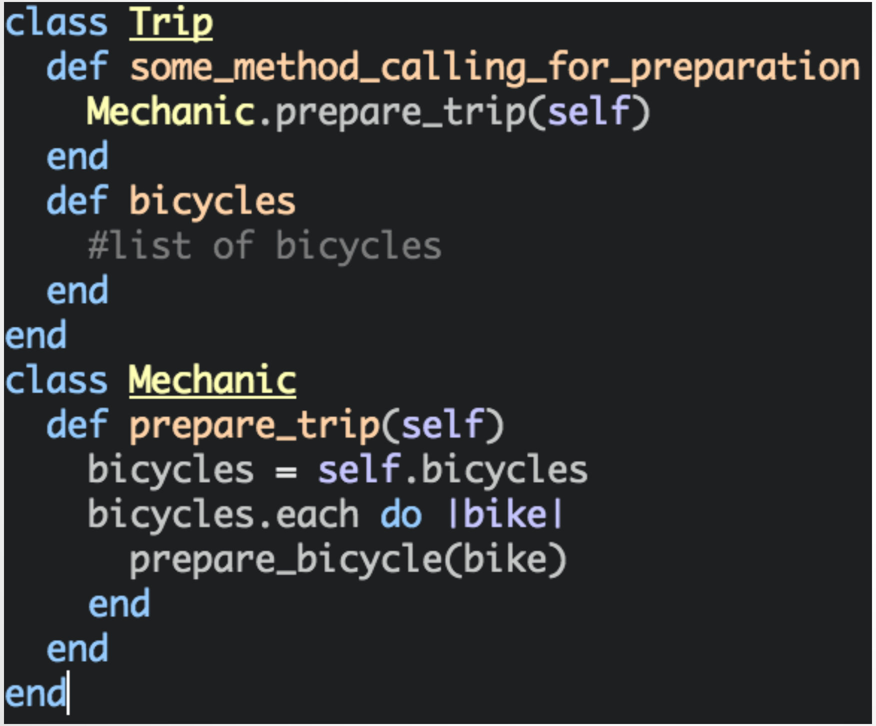


```
class Trip

  def bicycles(bicycles)
    bicycles.each do |bike|
      Mechanic.clean_bicycle(bike)
      Mechanic.pump_tires(bike)
      Mechanic.lube_chain(bike)
      Mechanic.check_brakes(bike)
    end
  end
end
```



```
class Trip

  def bicycles(bicycles)
    bicycles.each do |bike|
      Mechanic.prepare_bicycle(bike)
    end
  end
end

class Mechanic
  def prepare_bicycle(bike)
    clean_bicycle(bike)
    pump_tires(bike)
    lube_chain(bike)
    check_brakes(bike)
  end
end
```

Creating Flexible Interfaces



```
class Trip
  def some_method_calling_for_preparation
    Mechanic.prepare_trip(self)
  end
  def bicycles
    #list of bicycles
  end
end
class Mechanic
  def prepare_trip(self)
    bicycles = self.bicycles
    bicycles.each do |bike|
      prepare_bicycle(bike)
    end
  end
end
```

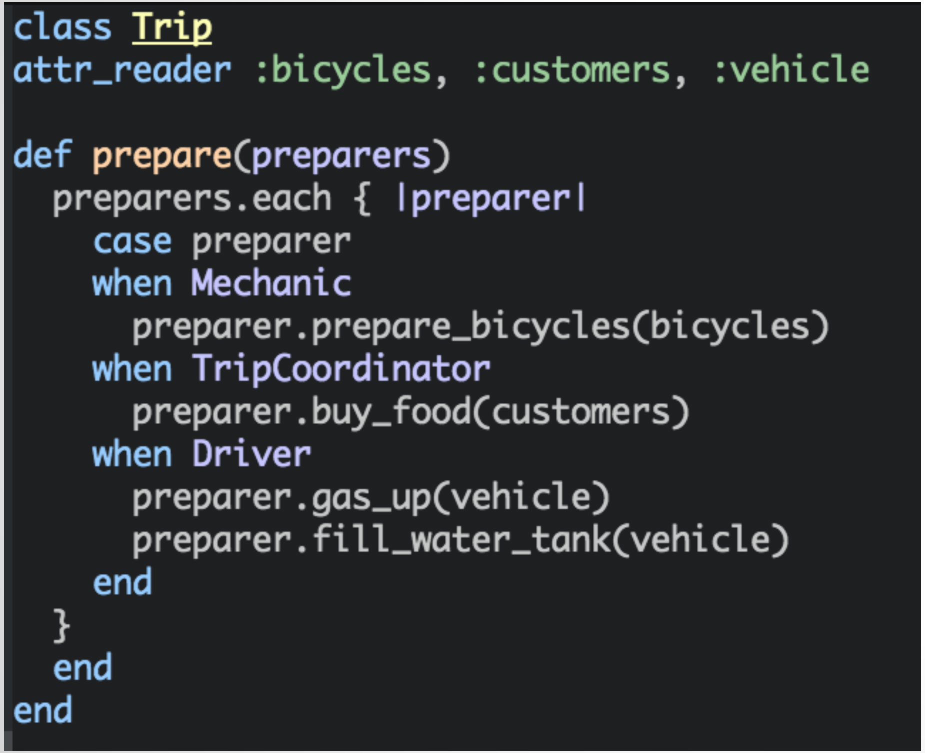
Creating Flexible Interfaces

Law Of Demeter

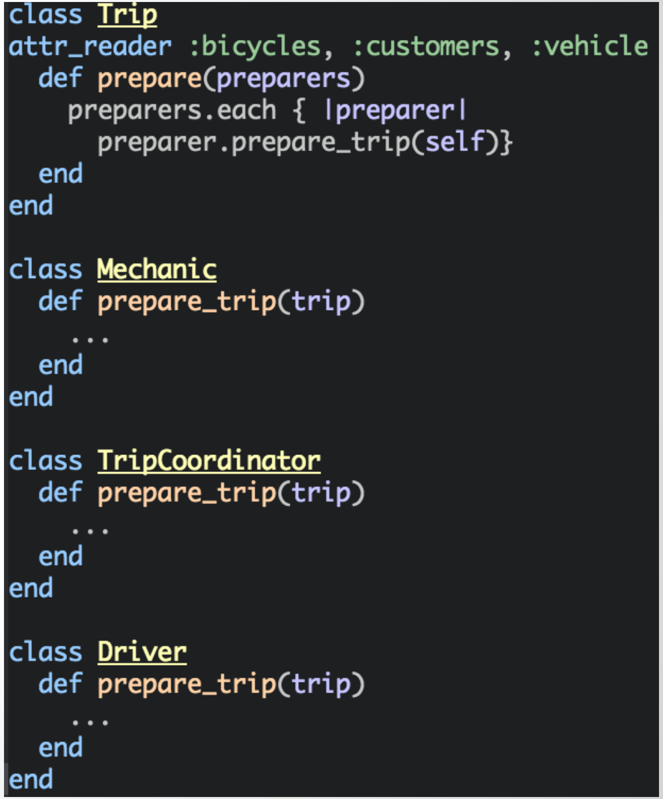
- Only talk to your immediate neighbors
- Use only one dot
    - `customer.bicycle.wheel.tire`
    - `customer.bicycle.wheel.rotate`
    - `hash.keys.sort.join(‘, ‘)`

Duck Typing

Duck Typing



```
class Trip
attr_reader :bicycles, :customers, :vehicle

def prepare(preparers)
  preparers.each { |preparer|
    case preparer
    when Mechanic
      preparer.prepare_bicycles(bicycles)
    when TripCoordinator
      preparer.buy_food(customers)
    when Driver
      preparer.gas_up(vehicle)
      preparer.fill_water_tank(vehicle)
    end
  }
  end
end
```



```
class Trip
attr_reader :bicycles, :customers, :vehicle
  def prepare(preparers)
    preparers.each { |preparer|
      preparer.prepare_trip(self)}
  end
end

class Mechanic
  def prepare_trip(trip)
    ...
  end
end

class TripCoordinator
  def prepare_trip(trip)
    ...
  end
end

class Driver
  def prepare_trip(trip)
    ...
  end
end
```

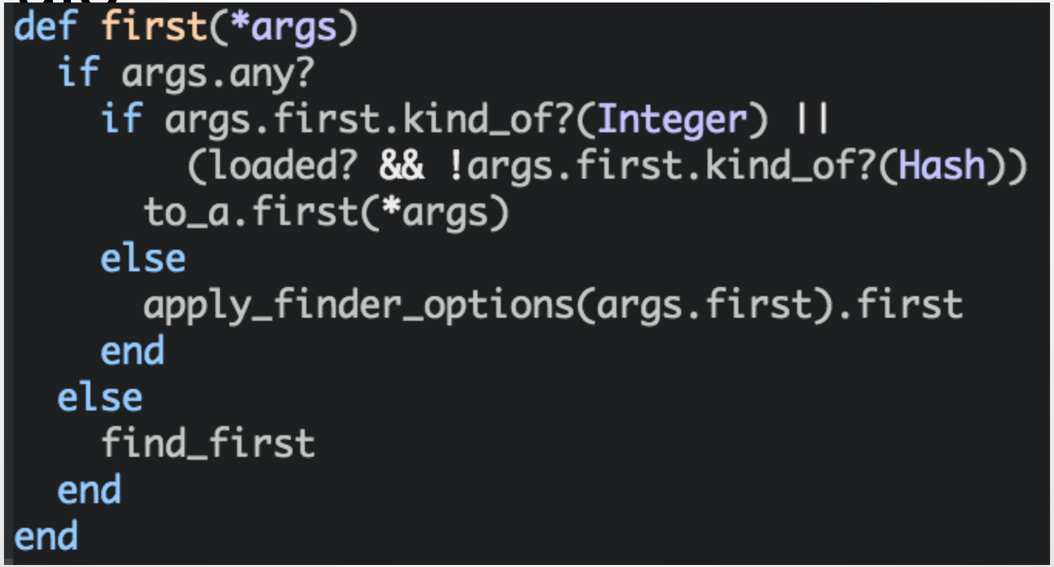
Duck Typing

Recognizing Hidden Ducks

- Case Statements that switch on Class
- `kind_of?` and `is_a?`
- `responds_to?`

Duck Typing

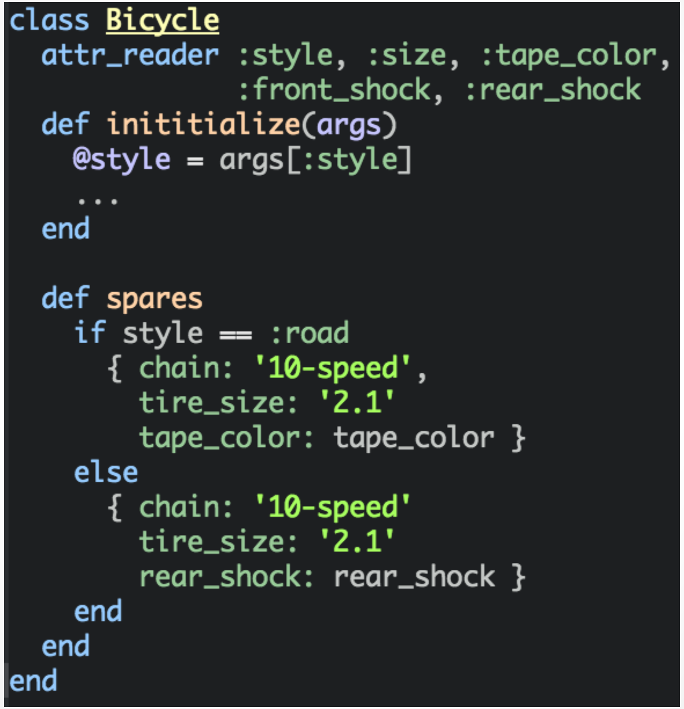
Just like all code, Duck Typing is not a hard and fast rule



```
def first(*args)
  if args.any?
    if args.first.kind_of?(Integer) ||
        (loaded? && !args.first.kind_of?(Hash))
      to_a.first(*args)
    else
      apply_finder_options(args.first).first
    end
  else
    find_first
  end
end
```

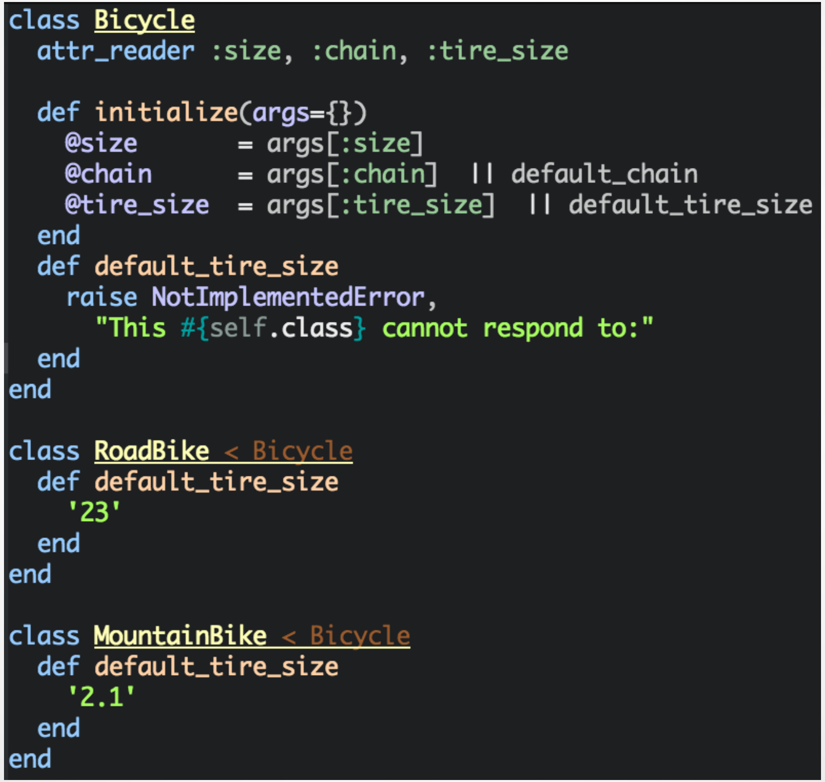
Acquiring Behavior Through Inheritance

Acquiring Behavior Through Inheritance



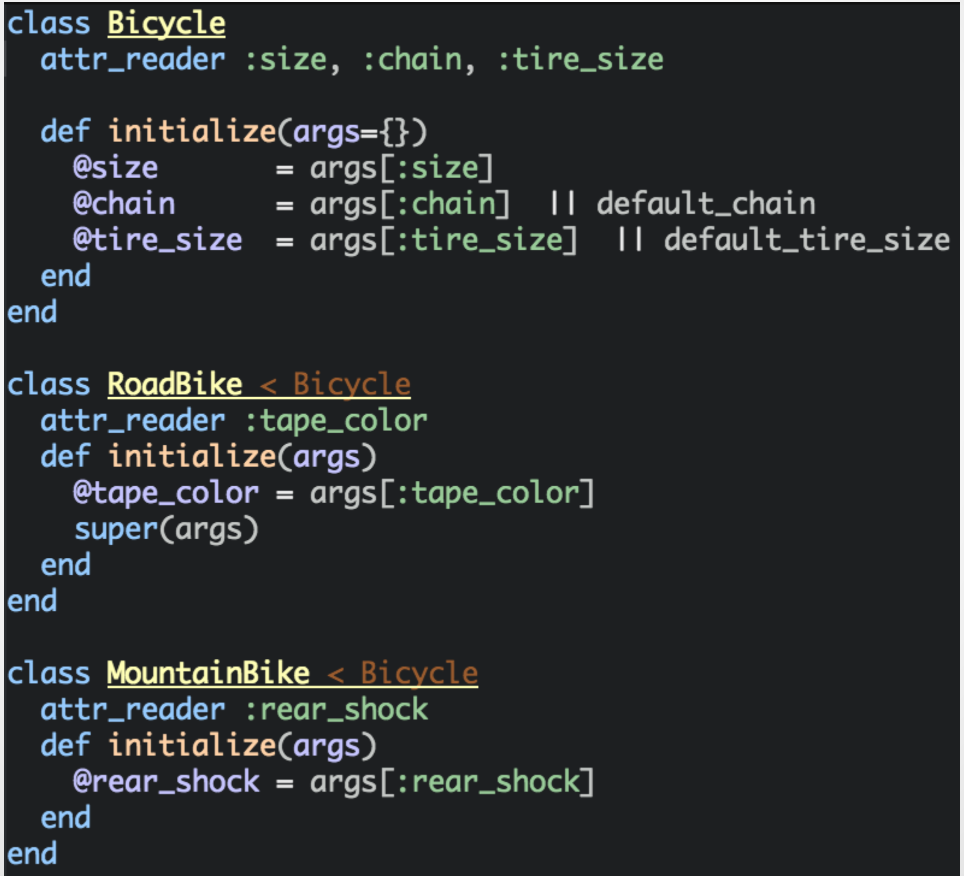
```ruby
class Bicycle
  attr_reader :style, :size, :tape_color,
              :front_shock, :rear_shock
  def inititialize(args)
    @style = args[:style]
    ...
  end

  def spares
    if style == :road
      { chain: '10-speed',
        tire_size: '2.1'
        tape_color: tape_color }
    else
      { chain: '10-speed'
        tire_size: '2.1'
        rear_shock: rear_shock }
    end
  end
end
```

Acquiring Behavior Through Inheritance



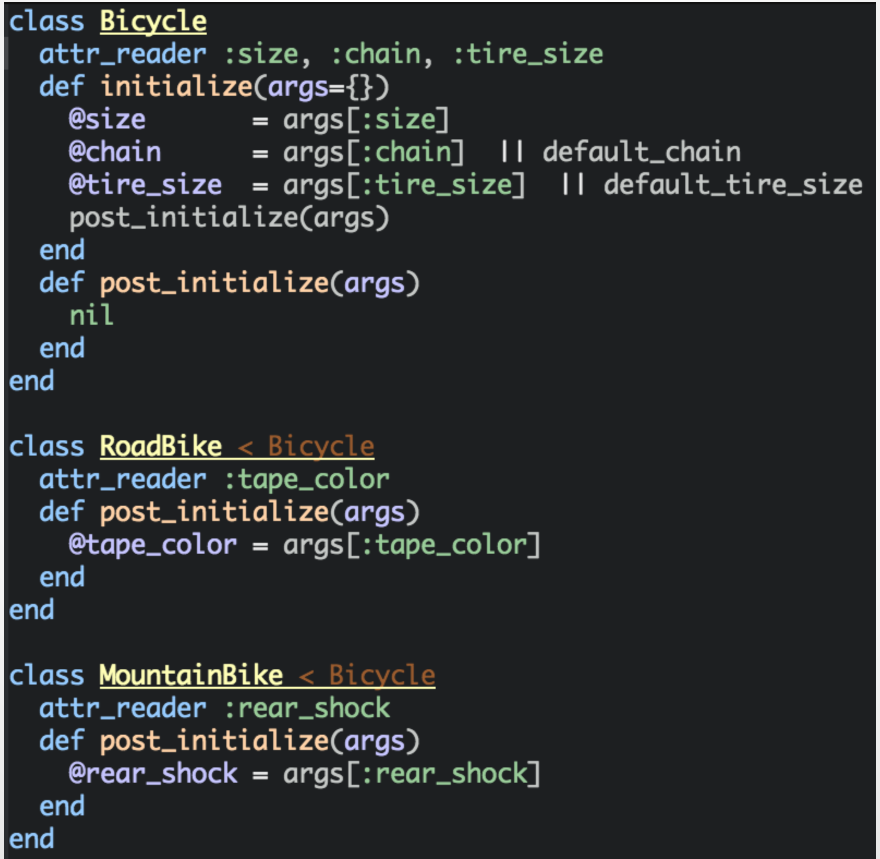
```ruby
class Bicycle
  attr_reader :size, :chain, :tire_size

  def initialize(args={})
    @size       = args[:size]
    @chain      = args[:chain]  || default_chain
    @tire_size  = args[:tire_size]  || default_tire_size
  end
  def default_tire_size
    raise NotImplementedError,
      "This #{self.class} cannot respond to:"
  end
end

class RoadBike < Bicycle
  def default_tire_size
    '23'
  end
end

class MountainBike < Bicycle
  def default_tire_size
    '2.1'
  end
end
```

Acquiring Behavior Through Inheritance



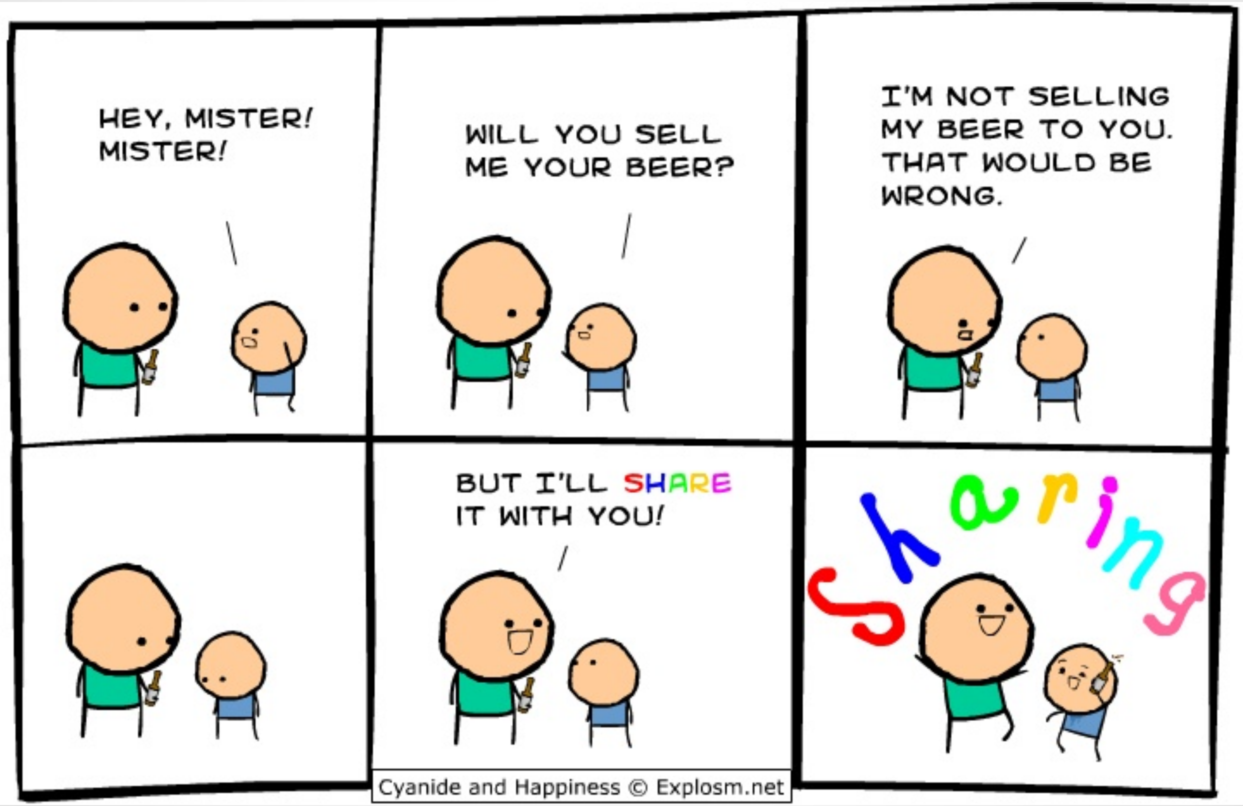
```
class Bicycle
  attr_reader :size, :chain, :tire_size

  def initialize(args={})
    @size       = args[:size]
    @chain      = args[:chain]  || default_chain
    @tire_size  = args[:tire_size]  || default_tire_size
  end
end

class RoadBike < Bicycle
  attr_reader :tape_color
  def initialize(args)
    @tape_color = args[:tape_color]
    super(args)
  end
end

class MountainBike < Bicycle
  attr_reader :rear_shock
  def initialize(args)
    @rear_shock = args[:rear_shock]
  end
end
```

Acquiring Behavior Through Inheritance

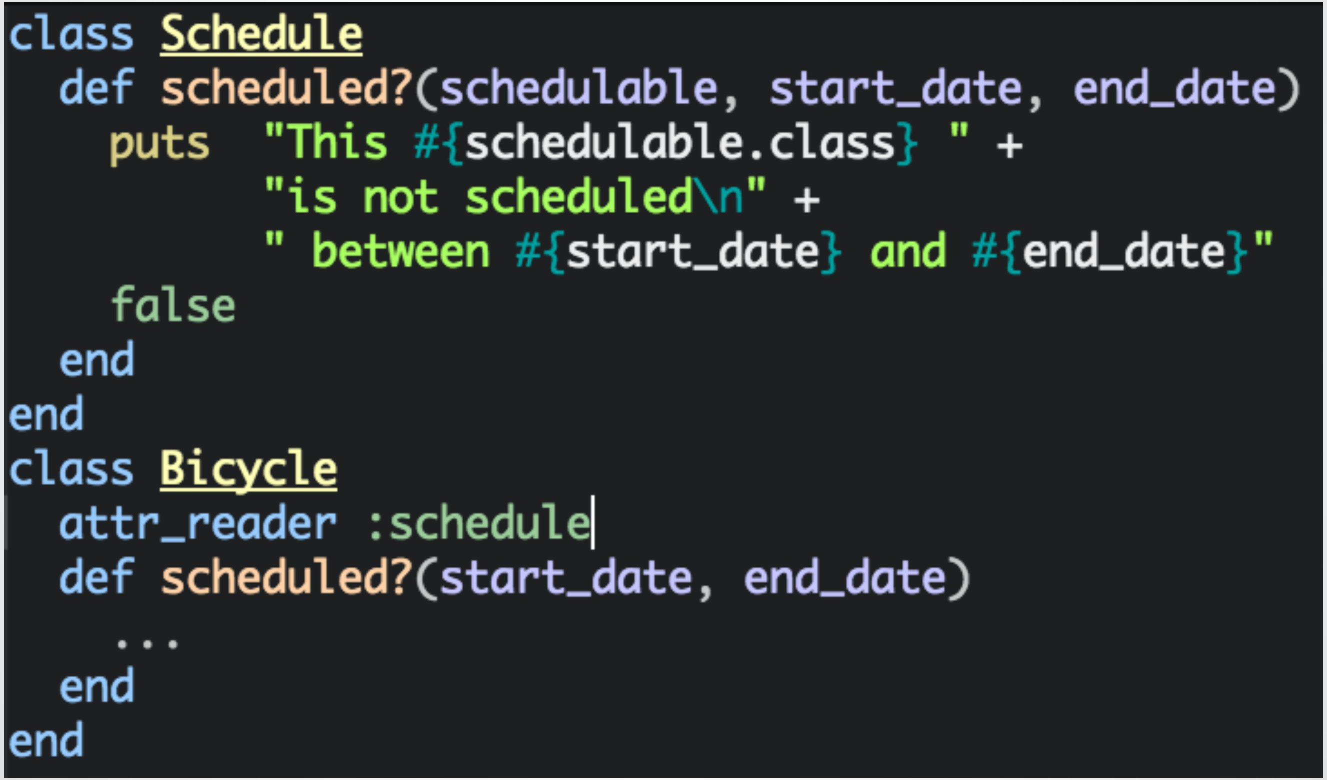


```ruby
class Bicycle
  attr_reader :size, :chain, :tire_size
  def initialize(args={})
    @size       = args[:size]
    @chain      = args[:chain]  || default_chain
    @tire_size  = args[:tire_size]  || default_tire_size
    post_initialize(args)
  end
  def post_initialize(args)
    nil
  end
end

class RoadBike < Bicycle
  attr_reader :tape_color
  def post_initialize(args)
    @tape_color = args[:tape_color]
  end
end

class MountainBike < Bicycle
  attr_reader :rear_shock
  def post_initialize(args)
    @rear_shock = args[:rear_shock]
  end
end
```

Sharing Role Behavior With Modules

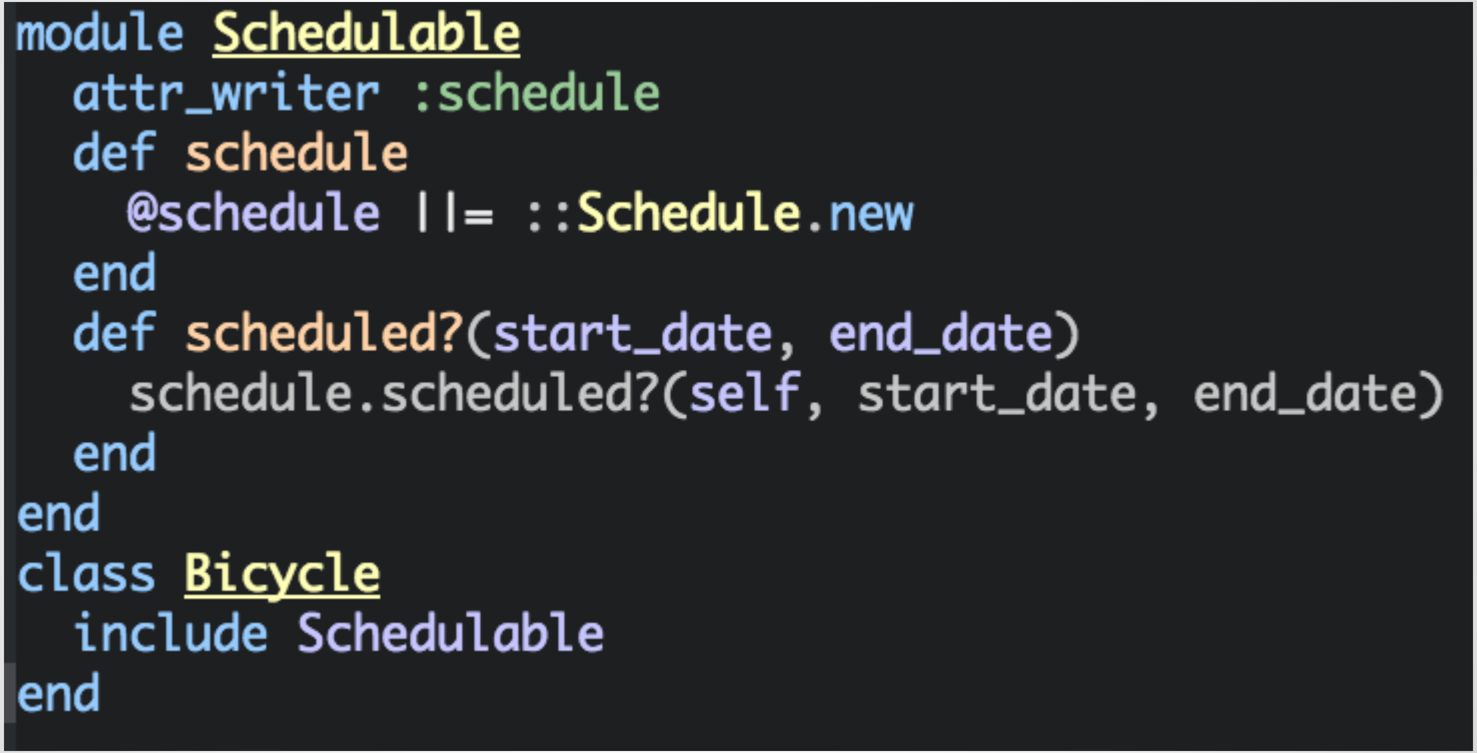

HEY, MISTER! MISTER!
WILL YOU SELL ME YOUR BEER?
I'M NOT SELLING MY BEER TO YOU. THAT WOULD BE WRONG.
BUT I'LL SHARE IT WITH YOU!
Sharing
Cyanide and Happiness © Explosm.net

Sharing Role Behavior With Modules



```ruby
class Schedule
  def scheduled?(schedulable, start_date, end_date)
    puts  "This #{schedulable.class} " +
          "is not scheduled\n" +
          " between #{start_date} and #{end_date}"
    false
  end
end
class Bicycle
  attr_reader :schedule
  def scheduled?(start_date, end_date)
    ...
  end
end
```

Sharing Role Behavior with Modules



```ruby
module Schedulable
  attr_writer :schedule
  def schedule
    @schedule ||= ::Schedule.new
  end
  def scheduled?(start_date, end_date)
    schedule.scheduled?(self, start_date, end_date)
  end
end
class Bicycle
  include Schedulable
end
```

Sharing Role Behavior with Modules

Object

Modules Defined in Bicycle

Bicycle

Modules In MountainBike

MountainBike

Methods defined in modules extended by this instance of mountain bike

Singleton Class

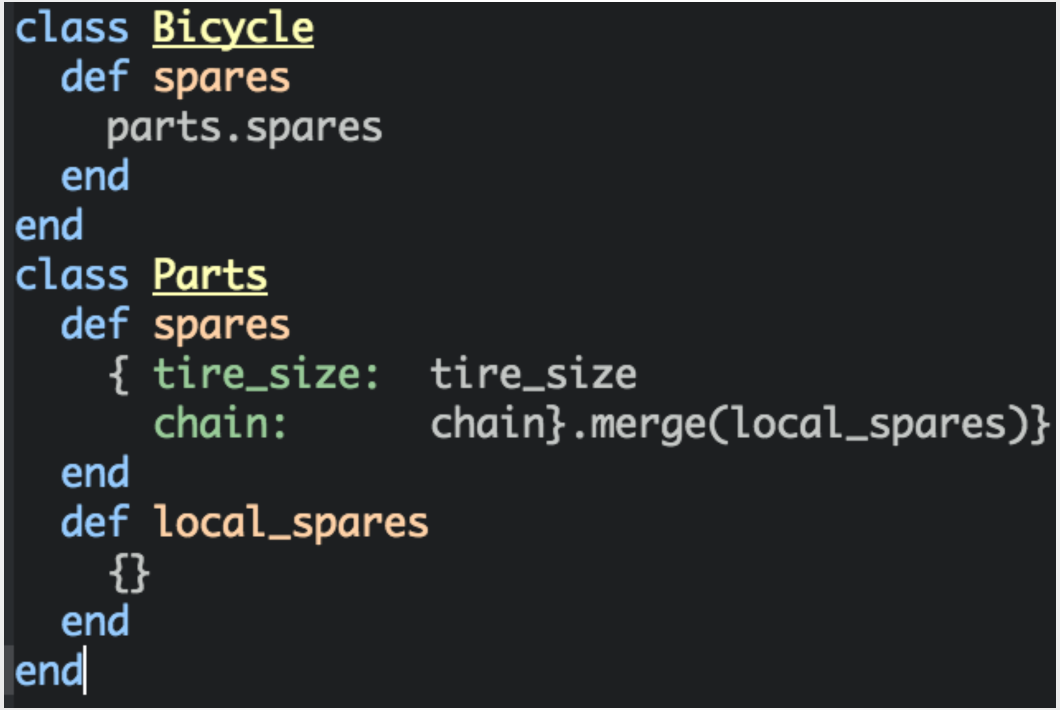
Sharing Role Behavior With Modules

Writing Inheritable Code

- Insist on the Abstraction
  - Superclass code should apply to every class that inherits it
- Liskov Substitution Principle
  - Every subclass should be substitutable for its superclass

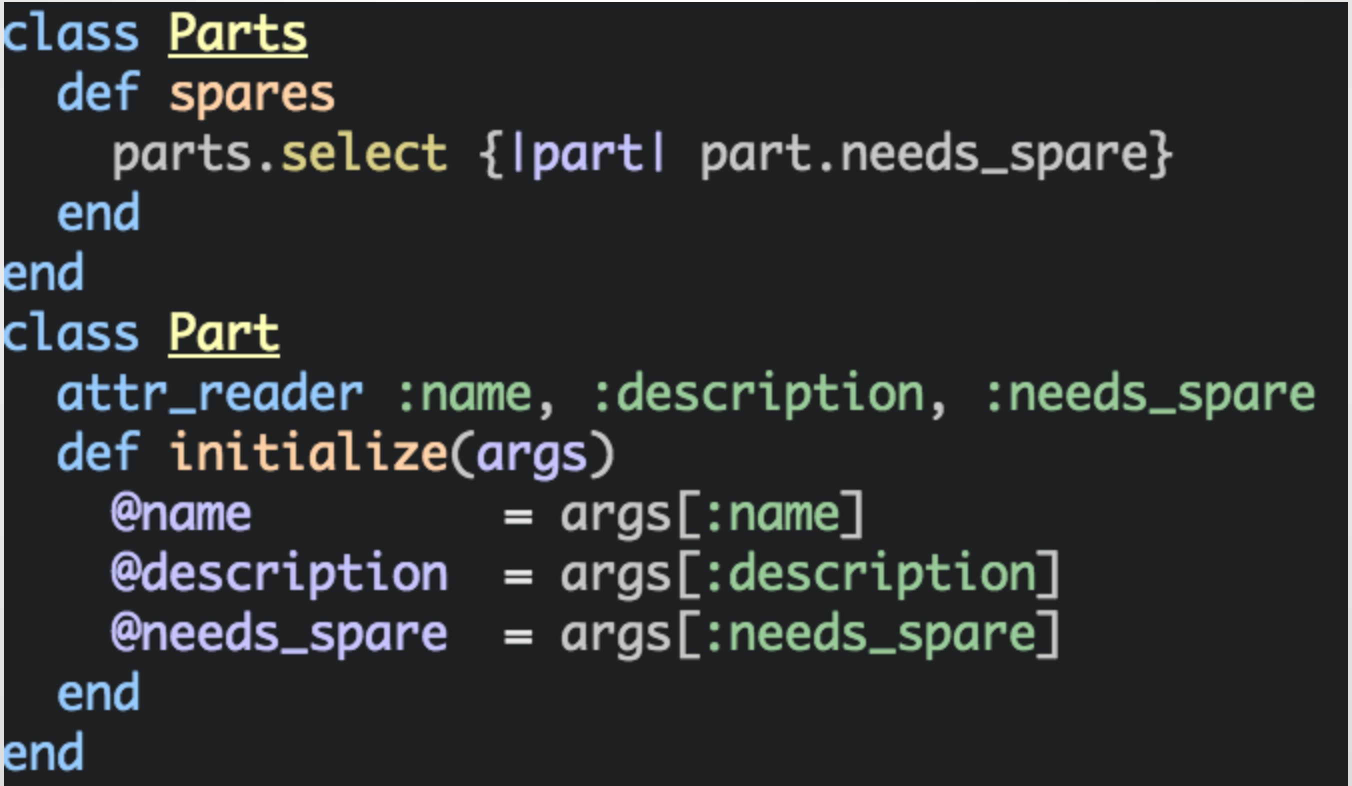
Combining Objects with Composition

Combining Objects with Composition



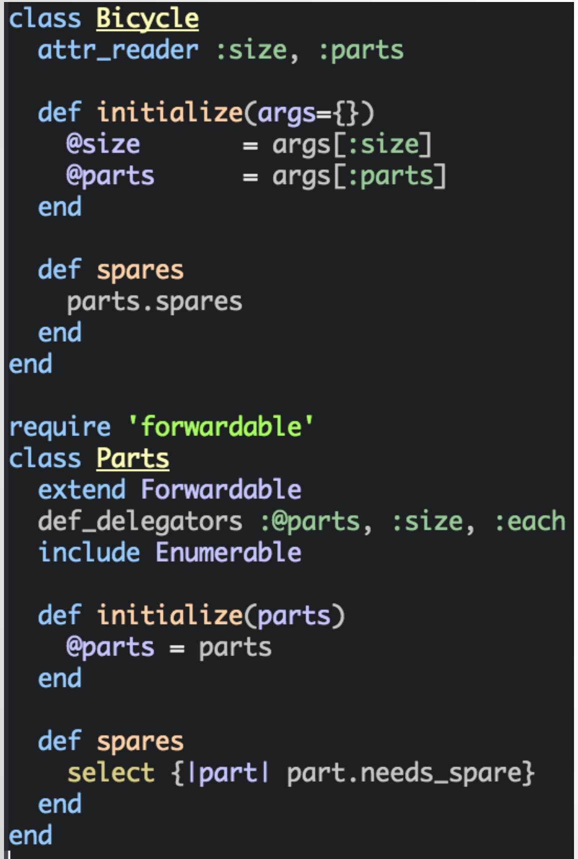
```
class Bicycle
  def spares
    parts.spares
  end
end
class Parts
  def spares
    { tire_size:  tire_size
      chain:      chain}.merge(local_spares)}
  end
  def local_spares
    {}
  end
end
```

Combining Objects with Composition

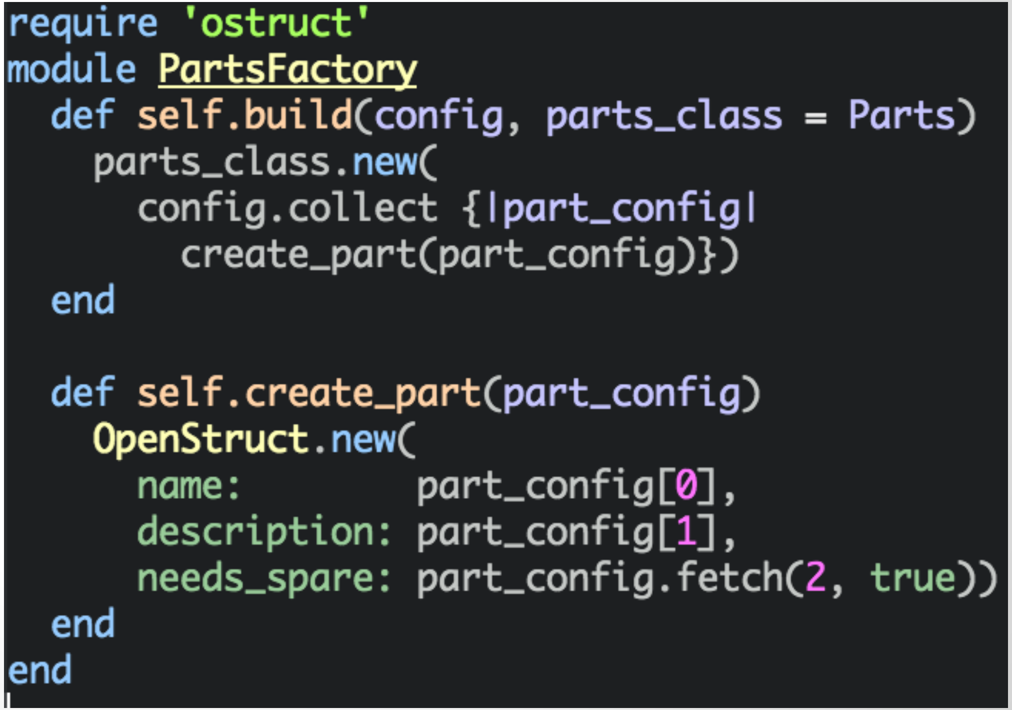


```
class Parts
  def spares
    parts.select {|part| part.needs_spare}
  end
end
class Part
  attr_reader :name, :description, :needs_spare
  def initialize(args)
    @name         = args[:name]
    @description  = args[:description]
    @needs_spare  = args[:needs_spare]
  end
end
```

Combining Objects with Composition



```ruby
class Bicycle
  attr_reader :size, :parts

  def initialize(args={})
    @size       = args[:size]
    @parts      = args[:parts]
  end

  def spares
    parts.spares
  end
end

require 'forwardable'
class Parts
  extend Forwardable
  def_delegators :@parts, :size, :each
  include Enumerable

  def initialize(parts)
    @parts = parts
  end

  def spares
    select {|part| part.needs_spare}
  end
end
```



```ruby
require 'ostruct'
module PartsFactory
  def self.build(config, parts_class = Parts)
    parts_class.new(
      config.collect {|part_config|
        create_part(part_config)})
  end

  def self.create_part(part_config)
    OpenStruct.new(
      name:        part_config[0],
      description: part_config[1],
      needs_spare: part_config.fetch(2, true))
  end
end
```

Combining Objects with Composition

Composition vs. Inheritance

- Faced with a problem that can be solved by composition, you should be biased in favor of it
- Contains fewer built-in dependencies

Combining Objects with Composition

Benefits of Inheritance

- Reasonable
- Usable; Create Subclass with no change to existing code
- Exemplary

Costs of Inheritance

- Can be used for wrong problems
- May be used for purposes you did not anticipate

Combining Objects with Composition

Benefits of Composition

- Transparent Parts
- Reasonable
- Usable

Costs of Composition

- Opaque Whole
- No message delegation; Objects must explicitly know which messages to delegate to whom

Combining Objects with Composition

General Rules:

- Use Inheritance for has-a
- Use duck-types for behaves-like-a
- Composition for has-a

Summing Up

- Object Oriented Code is About Messages
- Send the Simplest Messages Allowable
- Decrease Dependencies by Streamlining Interfaces
- Use Inheritance, Composition, and Modularization to DRY out code