



# Object Oriented Programming

## OOP Paradigm

Mental Model

The core concepts are:

Relationships

## Classes & Objects

What is an object ?

What is not an object ?

The class-object relationship

Relationships

In classes, you will find the attributes and behaviors of objects:

**Instantiation: the act of creating a new object**

## Encapsulation

What is encapsulation ?

Abstraction vs encapsulation

## Abstraction

## Inheritance

Importance

Subclass < Superclass

Use the < symbol

Module

Keyword `super`

## Polymorphism

### **Purpose**

In relation to inheritance

Through Duck Typing

Through the use of `Module`

Instance: Variable, Method, Scope, Inheritance, PPP, `to_s`

Instance Variables

Instance Variable Scope

With Inheritance

Instance Methods

Accessor Methods/ Getter and Setter

Rubyist Convention for getter and setter methods

`attr*`

**Using `attr_writer` method requirement**

Public, Private, Protected

Differences between `private` and `protected`

The `to_s` Method

Class: Variables, Methods, Scope, Inheritance

Class Variables

Class Variable Scope

With Inheritance

Class Methods

Class Constants: Scope, Inheritance, Lookup

Class Constants

Constant Variable Scope

With Inheritance

self

Instance method

class
Using <code>self</code>
Module
Main purpose: Supporting multiple inheritance
Namespacing
Mixing in Modules
Tip
Method Lookup Path
Method Lookup Path
We can use <code>ancestors</code> methods on any class to find out the method lookup chain
Fake Operator
Methods Vs Operators
Fake Operator: Comparison Method <code>#&gt;</code>
Fake Operator: Shift Method <code>#&lt;&lt;</code>
Fake Operator: The plus method
Fake Operators: Element setter and getter methods
General Design Consideration
Equivalence
Comparison in Ruby
Ruby's Equality Instance Methods
<code>==</code>
<code>===</code>
<code>eq?</code>
<code>equal?</code>
<code>object_id</code>
Design Considerations
Collaborator Objects
Objects, state and instance variables
Term Collaborator Object
Design Considerations

# OOP Paradigm

## Mental Model

Sub atomic atoms make up atoms. Atoms make up the world... Humans, when alive, have behaviors. Humans can also have other objects and relationships with other humans.

OOP is seeing the world through these connections, state, interactions and modelling the design of the software through these interactions. Both for composition and behavior.

## The core concepts are:

1. **Abstraction** is **simplifying** complex reality by **modeling** classes appropriate to the problem.
2. **Polymorphism** is the process of using an operator or **method** in different ways for different data input.
3. **Encapsulation** hides the **implementation**
4. **Inheritance** is a way to form **new** classes using classes that have already been defined.

## Relationships

Objects and classes are at the core of OOP. Once cannot exist without the other.

## OO vs Procedural

Aa Benefits	≡ Procedural
-------------	--------------

Aa Benefits	☰ Procedural
<u>Aids in compartmentalizing our program with clear boundaries</u>	Can be seen as one large monolith that is deeply interconnected
<u>Many independent sub-programs that are interact</u>	One program, cannot section out parts
<u>Easier to maintain and update since compartementalized</u>	Many dependencies leading to many changes for an update
<u>A change doesn't affect the entire program</u>	A change most likely affects many parts of the program
<u>Code is useful in many contexts</u>	Code is useful only in one context
<u>Leverages OOP concepts for re-usability, security and flexibility.</u>	Code is as is, cannot be extended or reused
<u>Offers a concrete and systematic way to model the problem, nous represent objects, classes are the template for objects, OOP concepts help build abstraction</u>	Better for smaller, concrete and sequential problems
<u>Creating objects allows programmers to think more abstractly about the code they are writing.</u>	
<u>Objects are represented by nouns so are easier to conceptualize.</u>	
<u>It allows us to only expose functionality to the parts of code that need it, meaning namespace issues are much harder to come across.</u>	
<u>It allows us to easily give functionality to different parts of an application without duplication.</u>	
<u>We can build applications faster as we can reuse pre-written code.</u>	
<u>As the software becomes more complex this complexity can be more easily managed.</u>	

## Classes & Objects

### What is an object ?

Anything that can be said to have a **value** is an **object**: that includes numbers, strings, arrays, and even classes and modules.

### What is not an object ?

- **methods** and **blocks** are two that stand out.
- **keywords** are another.

### The class-object relationship

Objects are created from classes. The **class** is the **mold**, and the object is the thing you produce out of the mold.

*The class defines the instance variables (which we might think of as attributes), so all instances of the class have the same attributes, but can have different state.*



A class can only inherit from **one** other class.

#### ▼ Two different objects from the same class

```
"hello".class # String
"world".class # String
```

We use the `class` instance method to determine what is the class of the object it's called on. Here, both objects are instantiated from the `String` Class.

## Relationships

- Everything revolves around the objects in OOP.
- Their capacity to interact with methods, other objects and to respond back enables us to create systems for transformation and communication.
- All this is done through different techniques, with proper read and write access.
- The main concepts are a way to model and implement OOP.

## In classes, you will find the attributes and behaviors of objects:

*A class is like a blueprint of what an object should be made of and what it should do.*

▼ **States:** This is to track attributes for an individual object.

- We use **instance variables** to track this information. They also have their own scope (object or instance level Instance: Variable, Method, Scope, Inheritance, PPP, to\_s)
- We can have different instances of the same class with different state, but they will share common methods.
- Instance variables keep track of state.

▼ **Behaviors:** This is what an object is capable of doing.

- Behaviors defined as instance methods in a class are available to objects (or instances) of that class.
- Instance methods expose behavior for objects.

## Instantiation: the act of creating a new object

▼ **Example: Creating a class**

```
class Cat
end
```

Defining a class is similar to defining a method, except for two notable differences:

- When defining a class, the reserved word `class` is used instead of `def`.
- When naming a class, the `CamelCase` format is used instead of the `snake_case` format.

Classes, like methods, also use the reserved word `end` to finish the definition.

▼ **Instantiation of a new object from a class**

filename: `snake_case.rb` that reflects the class name.

```
class GoodDog
end

sparky = GoodDog.new
```

We first define a class, `GoodDog`. Then, the class method invocation `new` is called on the class. This **instantiates** a new `GoodDog` object which the local variable `sparky` is now referencing.

Using the `.new` after the class name will tell Ruby this new object is an instance of `GoodDog`.

▼ **Example: Instantiating a new object with the `initialize` method without instance variables**

```
class GoodDog
  def initialize
    puts "This object was initialized!"
  end
end

sparky = GoodDog.new      # => "This object was initialized!"
```

The `initialize` method gets called every time you create a new object. Calling the `new` class method eventually leads us to the `initialize` instance method.

In the above example, instantiating a new `GoodDog` object triggered the `initialize` method and resulted in the string being outputted. We refer to the `initialize` method as a **constructor**, because it gets triggered **whenever** we create a new object.

#### ▼ Example: Instantiating a new object with the `initialize` method with instance variable

Through the `new` method, you pass arguments into the `initialize` method.

```
class GoodDog
  def initialize(name)
    @name = name
  end
end

sparky = GoodDog.new("Sparky")
```

Here, the string "Sparky" is being passed from the `new` method through to the `initialize` method, and is assigned to the local variable `name`. Within the constructor (i.e., the `initialize` method), we then set the instance variable `@name` to `name`, which results in assigning the string "Sparky" to the `@name` instance variable.

The instance variable is responsible for keeping track of information about the state of an object. The name of the `sparky` object is the string "Sparky".

The state for the object is tracked in the instance variable `@name`.

If we wanted, we could create more `GoodDog` objects. Every object's state is unique, and instance variables are how we keep track.

## Encapsulation

### What is encapsulation ?

Encapsulation is a technique used to hide the **implementation** details of a class from other objects. It provides a form of data protection.

#### In action:

1. Variables: A label for an object/Data Structure
2. Object: Bundling attributes (state) via instance variables
3. Methods: Logic
4. Class: Methods, Constants, others

#### How:

1. Creating a single unit for reference *see above*
2. Hiding state and behavior from direct access (user must use public interface to access what is available)

3. Using `private` to hide data from the public interface
4. Using `protected` to allow for inter-class interactions between state and behavior, but not wanting the information on either's public interface

#### Benefits:

1. The data cannot be changed or manipulated without obvious intention
2. It creates boundaries in an application by hiding pieces of functionality and not making it available to the rest of the code base.
3. The user only has access to the public interface to access state and methods
- ▼ 4. Adds another layer of abstraction

By thinking of objects in real world nouns, we can associate verbs to the objects. The process of solving a problem then becomes more concrete. This simplifies the process of thinking in an OO way.

## Abstraction vs encapsulation

Where abstraction aims to expose the essence of a subject, encapsulation aims to hide extra details.

#### ▼ Example of Encapsulation

```
class Dog
  attr_reader :nickname

  def initialize(n)
    @nickname = n
  end

  def change_nickname(n)
    self.nickname = n # can also be written @nickname = n
  end

  def greeting
    "#{nickname.capitalize} says Woof Woof!"
  end

  private
  attr_writer :nickname
end

dog = Dog.new("rex")
dog.change_nickname("barny") # changed nickname to "barny"
puts dog.greeting # Displays: Barny says Woof Woof!
```

In this example, we can change the nickname of a dog by calling the `change_nickname` method without needing to know how the `Dog` class and this method are implemented.

The same thing happens when we call the method `greeting` on a `Dog` object. The output is `Barny says Woof Woof!`, with the dog's nickname capitalized. Again, we don't need to know how the method is implemented. The main point is that we expect a greeting message from the dog and that's what we get.

Note that the `setter` method for `nickname` is private: it is not available outside of the class and `dog.nickname = "barny"` would raise an error.



On a final note, always keep in mind that the class should have as few public methods as possible. It lets us simplify using that class and protect data from undesired changes from the outer world.

## Abstraction

**Abstraction** is simplifying complex reality by **modeling** classes appropriate to the problem. This is known as **design-level**.

**Design-Level:** High-level, 'birds eye' view of a program. The realm of abstraction, responsible for getting rid of unnecessary data that doesn't relate to the overall design of a program.

## Inheritance

Inheritance is used to create relationships and model hierarchical connections between certain components of our program.



If there's an **"is-a" relationship**, **class inheritance** is usually the correct choice.  
If there's a **"has-a" relationship**, **interface inheritance** is generally a better choice.  
For example, a **dog** "is an" **animal** and it "has an" ability to **swim**.

### Importance

- In **design**, it helps **DRY** up your code.
- **Class based inheritance** works great when it's used to model hierarchical domains.
  - Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.
- Ruby does not support multiple levels of inheritances but Ruby supports **mixins**.
  - A mixin is like a specialized implementation of multiple inheritance in which only the interface portion is inherited.
  - We can mix in as many modules as we want into classes.

### Subclass < Superclass

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class.

This existing class is called the **base class or superclass**, and the new class is referred to as the **derived class or sub-class**.

**Classes inherit from other classes;** objects do not inherit from their class, rather the class acts as a mold or template for objects of that class and determines the attributes and behaviors of those objects.

- A **superclass** can be written with basic and large reusability.
- A **subclass** inherits the behaviors from the **superclass** and has it's own fine-grained, detailed behavior.
- A class can only inherit from 1 parent class.

### Use the < symbol

to signify that the a **subclass** is **inheriting** from the **superclass** :

#### ▼ Example 1

superclass `Animal`

```
class Animal
  def speak
    "Hello!"
  end
end
```

```

class GoodDog < Animal
end

class Cat < Animal
end

sparky = GoodDog.new
paws = Cat.new
puts sparky.speak      # => Hello!
puts paws.speak        # => Hello!

```

Here, we're extracting the `speak` method to a superclass `Animal`, and we use inheritance to make that behavior available to `GoodDog` and `Cat` classes.

This means that all of the methods in the `Animal` class are available to the `GoodDog` class for use. We also created a new class called `Cat` that inherits from `Animal` as well.

### ▼ Example 2

superclass `Vehicule`

```

class Vehicle
  def self.gas_mileage(gallons, miles)
    puts "#{miles / gallons} miles per gallon of gas"
  end
end

class MyCar < Vehicle
  NUMBER_OF_DOORS = 4
  #code omitted for brevity...
end

class MyTruck < Vehicle
  NUMBER_OF_DOORS = 2
end

```

A subclass can only have 1 superclass.

### ▼ Example 3

Superclass `Pet` and `Dog`

```

class Pet
  def run
    'running!'
  end

  def jump
    'jumping!'
  end
end

class Dog < Pet
  def speak
    'bark!'
  end

  def fetch
    'fetching!'
  end

  def swim
    'swimming!'
  end
end

class Cat < Pet
  def speak

```

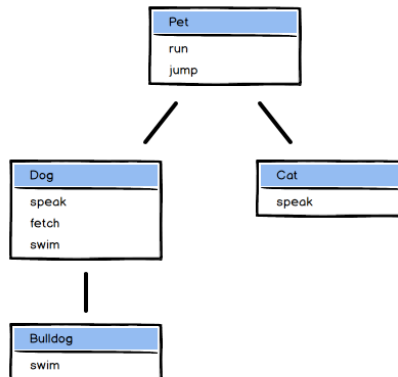


```

    'meow!'
  end
end

class Bulldog < Dog
  def swim
    "can't swim!"
  end
end
end

```



## Module

You **cannot instantiate from a modules** (i.e., no object can be created from a module).

Modules are used only for namespacing and grouping common methods together.

### ▼ Fundamentals of Modules

▼ It's important to remember that every class you create inherently subclasses from class `Object`. The Object class is built into Ruby and comes with many critical methods.

```

class Parent
  def say_hi
    p "Hi from Parent."
  end
end

Parent.superclass # => Object

```

Methods defined in the `Object` class are available in **all classes**.

▼ With inheritance, a subclass can override a superclass's methods

```

class Child < Parent
  def say_hi
    p "Hi from Child."
  end
end

child = Child.new
child.say_hi # => "Hi from Child."

```

This means that, if you accidentally override a method that was originally defined in the `Object` class, it can have far-reaching effects on your code. For example, `send` is an instance method that all classes inherit from `Object`. If you defined a new `send` instance method in your class, all objects of your class will call your custom `send` method, instead of the one in class `Object`, which is probably the one they mean to call.

▼ Object `send` serves as a way to call a method by passing it a symbol or a string which represents the method you want to call.

```
son = Child.new
son.send :say_hi      # => "Hi from Child."
```

▼ When we define a `send` method in our `Child` class and then try to invoke Object's `send` method

```
class Child
  def say_hi
    p "Hi from Child."
  end

  def send
    p "send from Child..."
  end
end

lad = Child.new
lad.send :say_hi

# ArgumentError: wrong number of arguments (1 for 0)
# from (pry):12:in `send'
```

▼ `Object#instance_of?`

What this handy method does is to return `true` if an object is an instance of a given class and `false` otherwise.

In this example, we're going to override it:

```
class Child
  # other methods omitted

  def instance_of?
    p "I am a fake instance"
  end
end

heir = Child.new
heir.instance_of? Child

# ArgumentError: wrong number of arguments (1 for 0)
# from (pry):22:in `instance_of?'
```

- *More at Module Section*

## Keyword `super`

Ruby provides us with a built-in function called `super` that allows us to call methods up the method lookup path.

When you call `super` from within a method, it will search the inheritance hierarchy for a method by the same name and then invoke it. It's a **keyword**.

### ▼ Example 1: Instance Method

```

class Animal
  def speak
    "Hello!"
  end
end

class GoodDog < Animal
  def speak
    super + " from GoodDog class"
  end
end

sparky = GoodDog.new
sparky.speak      # => "Hello! from GoodDog class"

```

In the above example, we've created a simple `Animal` class with a `speak` instance method. We then created `GoodDog` which subclasses `Animal` also with a `speak` instance method to override the inherited version. However, in the subclass' `speak` method we use `super` to invoke the `speak` method from the superclass, `Animal`, and then we extend the functionality by appending some text to the result.

#### ▼ Example 2: `super` with `initialize`

```

class Animal
  attr_accessor :name

  def initialize(name)
    @name = name
  end
end

class GoodDog < Animal
  def initialize(color)
    super
    @color = color
  end
end

bruno = GoodDog.new("brown")
# => #<GoodDog:0x007fb40b1e6718 @color="brown", @name="brown">

```

In this example, we're using `super` with no arguments. However, the `initialize` method, where `super` is being used, takes an argument and adds a new twist to how `super` is invoked. Here, in addition to the default behavior, `super` automatically forwards the arguments that were passed to the method from which `super` is called ( `initialize` method in `GoodDog` class). At this point, `super` will pass the `color` argument in the `initialize` defined in the subclass to that of the `Animal` superclass and invoke it. That explains the presence of `@name="brown"` when the `bruno` instance is created. Finally, the subclass' `initialize` continues to set the `@color` instance variable.

#### When to Use `super` vs `super()`

##### ▼ Example 3: `super` with arguments

When called with specific arguments, eg. `super(a, b)`, the specified arguments will be sent up the method lookup chain.

```

class BadDog < Animal
  def initialize(age, name)
    super(name)
    @age = age
  end
end

BadDog.new(2, "bear")      # => #<BadDog:0x007fb40b2beb68 @age=2, @name="bear">

```

`super` takes an argument and this argument is sent to the superclass.

In this example when a `BadDog` class is created, the passed in name argument ("bear") is passed to the superclass and set to the `@name` instance variable.

#### ▼ Example 4: `super` without arguments

```
class Vehicle
  def start_engine
    'Ready to go!'
  end
end

class Truck < Vehicle
  def start_engine(speed)
    super() + " Drive #{speed}, please!"
  end
end

truck1 = Truck.new
puts truck1.start_engine('fast')
```

As we know, `#super` invokes the method in the inheritance hierarchy with the same name as the method in the child. Therefore, `#start_engine` in `Vehicle` will be invoked if we call `#super` within `#start_engine` in `Truck`.

The tricky part here is that `#start_engine` in `Vehicle` doesn't accept any arguments. If we passed `speed` as an argument, we would get an error. To remedy this, we can invoke `#super` **with empty parentheses**, which means no arguments will be passed.

- `super()` for informing that there are no arguments
- `super(arg1, arg2, ...)` to choose what arguments you want to pass

## Polymorphism

### Purpose

- Polymorphism refers to the ability of different objects to respond in different ways to the same message (or method invocation).
- The ability for data to be represented as many different types.
- OOP gives us flexibility in using pre-written code for new purposes.
- Method overriding is one way in which Ruby implements polymorphism.

### In relation to inheritance

#### ▼ Example of Polymorphism through inheritance + method overriding

```
class Animal
  def eat
    # generic eat method
  end
end

class Fish < Animal
  def eat
    # eating specific to fish
  end
end

class Cat < Animal
```

```

def eat
  # eat implementation for cat
end

def feed_animal(animal)
  animal.eat
end

array_of_animals = [Animal.new, Fish.new, Cat.new]
array_of_animals.each do |animal|
  feed_animal(animal)
end

```

Every object in the array is a different animal, but the client code can treat them all as a generic `animal`, i.e., an object that can `eat`. Thus, the public interface lets us work with all of these types in the same way even though the implementations can be dramatically different. That is polymorphism in action.

### ▼ Example of Method Overriding through Inheritance

```

class Animal
  def speak
    "Hello!"
  end
end

class GoodDog < Animal
  attr_accessor :name

  def initialize(n)
    self.name = n
  end

  def speak
    "#{self.name} says arf!"
  end
end

class Cat < Animal
end

sparky = GoodDog.new("Sparky")
paws = Cat.new

puts sparky.speak      # => Sparky says arf!
puts paws.speak        # => Hello!

```

In the `GoodDog` class, we're overriding the `speak` method in the `Animal` class because Ruby checks the object's class first for the method before it looks in the superclass. That means when we wrote the code `sparky.speak`, it first looked at sparky's class, which is `GoodDog`. It found the `speak` method there and used it. When we wrote the code `paws.speak`, Ruby first looked at paws's class, which is `Cat`. It did not find a `speak` method there, so it continued to look in `Cat`'s superclass, `Animal`. It found a `speak` method in `Animal`, and used it.

## Through Duck Typing

Duck typing in Ruby doesn't concern itself with the class of an object; instead, **it concerns itself with what methods are available on the object**. If an object "quacks" like a duck, then we can treat it like a duck.

### ▼ Example of Polymorphism through duck typing

We can see that there is no inheritance in this example, but we still have polymorphism.

Each class must define a `prepare_wedding` method and implement it in its own way. If we must add another `preparer`, we can create another class and just implement `prepare_wedding` to perform the appropriate actions.

```

class Wedding
  attr_reader :guests, :flowers, :songs

  def prepare(preparers)
    preparers.each do |preparer|
      preparer.prepare_wedding(self)
    end
  end
end

class Chef
  def prepare_wedding(wedding)
    prepare_food(wedding.guests)
  end

  def prepare_food(guests)
    #implementation
  end
end

class Decorator
  def prepare_wedding(wedding)
    decorate_place(wedding.flowers)
  end

  def decorate_place(flowers)
    # implementation
  end
end

class Musician
  def prepare_wedding(wedding)
    prepare_performance(wedding.songs)
  end

  def prepare_performance(songs)
    #implementation
  end
end

```

## Through the use of `Module`

We can achieve polymorphism with the use of `Module`.

The `Module` cannot create objects however. It contains shared behavior that can be mixed in (**mixin**) with another class, using the `include` method invocation.

When the `Module` has been mixed in with the class, the methods defined in the `Module` are available to the class and its objects.

## Instance: Variable, Method, Scope, Inheritance, PPP, `to_s`

### Instance Variables

An instance variable starts with a `@`.



You can ask an instance of a class for a list of it's instance variables with `#instance_variables`

▼ In this example, `@name` is an instance variable. This instance variable exists as long as the object instance exists. It is used to tie data to the object.

```

class GoodDog
  def initialize(name)

```

```

    @name = name
  end
end

```

A `GoodDog` object's `name` attribute can be saved into a `@name` instance variable as a string.

Instance variables can hold any object, including data structures like array and hashes.

#### ▼ Technical Communication Example

A `Person` object's `name` attribute can be saved into a `@name` instance variable as a string.

```

class Person
  def initialize(name)
    @name = name
  end

  def name
    @name
  end
end

joe = Person.new("Joe")
joe.name # => "Joe"

```

- More on [instantiation/initialize](#)

## Instance Variable Scope

- Instance variables are variables that start with `@` and are scoped at the object level. They are used to track individual object state, and do not cross over between objects.

▼ **Example: we can use a `@name` variable to separate the state of Person objects.**

```

class Person
  def initialize(n)
    @name = n
  end
end

bob = Person.new('bob')
joe = Person.new('joe')

puts bob.inspect # => #<Person:0x007f9c830e5f70 @name="bob">
puts joe.inspect # => #<Person:0x007f9c830e5f20 @name="joe">

```

- Because the scope of instance variables is at the object level, this means that the instance variable is accessible in an object's instance methods, even if it's initialized outside of that instance method.

▼ **Example: Instance variable in object scope**

```

class Person
  def initialize(n)
    @name = n
  end

  def get_name
    @name # is the @name instance variable accessible here?
  end
end

bob = Person.new('bob')
bob.get_name # => "bob"

```

- Unlike local variables, instance variables are accessible within an instance method even if they are not initialized or passed in to the method. Remember, their scope is at the object level.
- If you try to reference an uninitialized local variable, you'd get a `NameError`. But if you try to reference an uninitialized instance variable, you get `nil`.

▼ **Example: Instance variable that is not yet initialized**

```
class Person
  def get_name
    @name # the @name instance variable is not initialized anywhere
  end
end

bob = Person.new
bob.get_name # => nil
```

## With Inheritance

Instance Variables behave the way we'd expect. The only thing to watch out for is to make sure the instance variable is initialized before referencing it.

▼ **Example: How sub-classing affects instance variables**

```
class Animal
  def initialize(name)
    @name = name
  end
end

class Dog < Animal
  def dog_name
    "bark! bark! #{@name} bark! bark!" # can @name be referenced here?
  end
end

teddy = Dog.new("Teddy")
puts teddy.dog_name # => bark! bark! Teddy bark! bark!
```

When we instantiated `teddy`, we called `Dog.new`. Since the `Dog` class doesn't have an `initialize` instance method, the method lookup path went to the super class, `Animal`, and executed `Animal#initialize`. That's when the `@name` instance variable was initialized, and that's why we can access it from `teddy.dog_name`.

▼ **Example: Without proper initialization**

```
class Animal
  def initialize(name)
    @name = name
  end
end

class Dog < Animal
  def initialize(name)
  end

  def dog_name
    "bark! bark! #{@name} bark! bark!" # can @name be referenced here?
  end
end

teddy = Dog.new("Teddy")
puts teddy.dog_name # => bark! bark! bark! bark!
```



In this case, `@name` is `nil`, because it was never initialized. The `Animal#initialize` method was never executed. Remember that uninitialized instance variables return `nil`.

#### ▼ Example: With modules

```
module Swim
  def enable_swimming
    @can_swim = true
  end
end

class Dog
  include Swim

  def swim
    "swimming!" if @can_swim
  end
end

teddy = Dog.new
teddy.swim # => nil
```

Since we didn't call the `Swim#enable_swimming` method, the `@can_swim` instance variable was never initialized. Assuming the same module and class from above, we need to do the following:

```
teddy = Dog.new
teddy.enable_swimming
teddy.swim # => swimming!
```

## Instance Methods

The instance methods are also defined in the same way as we define any other method using `def` keyword and they can be used using a class instance only as shown below. Their functionality is not limited to access the instance variables, but also they can do a lot more as per your requirement.

#### ▼ We can define methods in a class:

```
class GoodDog
  def initialize(name)
    @name = name
  end

  def speak # the return value of speak is "Arf!"
    "Arf!"
  end
end

sparky = GoodDog.new("Sparky")
sparky.speak # returns "Arf!"
```

#### ▼ All objects of the same class have the same behaviors, though they contain different states:

```
fido = GoodDog.new("Fido")
puts fido.speak # => Arf! is output
               # nil is returned
```

#### ▼ Instance methods have access to instance variables

```

def speak
  "#{@name} says arf!"
end

puts sparky.speak      # => "Sparky says arf!"
puts fido.speak        # => "Fido says arf!"

```

This exposes information about the state of the object using instance methods.

## Accessor Methods/ Getter and Setter

To make the instance variables available from outside the class, they must be defined within accessor methods, these accessor methods are also known as a getter methods.

### ▼ Getter method

In order to return a value stored in an instance variable (`@...`), we are required to create a method that will return the value in the instance variable. This is called a **getter method**.

```

class GoodDog
  def initialize(name)
    @name = name
  end

  def get_name
    @name
  end

  def speak
    "#{@name} says arf!"
  end
end

sparky = GoodDog.new("Sparky")
puts sparky.speak      # => Sparky says arf!
puts sparky.get_name   # => Sparky

```

### ▼ Setter method

To use the `set_name=` method normally, we would expect to do this: `sparky.set_name="Spartacus"`, where the entire `"set_name="` is the method name, and the string "Spartacus" is the argument being passed in to the method.

Ruby recognizes that this is a setter method and allows us to use the more natural assignment syntax: `sparky.set_name = "Spartacus"`. Just realize there's a method called `set_name=` working behind the scenes, and we're just seeing some Ruby syntactical sugar.

```

class GoodDog
  def initialize(name)
    @name = name
  end

  def get_name
    @name
  end

  def set_name=(name)
    @name = name
  end

  def speak
    "#{@name} says arf!"
  end
end

```

```

sparky = GoodDog.new("Sparky")
puts sparky.speak # Sparky says arf!
puts sparky.get_name # Sparky
sparky.set_name = "Spartacus"
puts sparky.get_name # Spartacus

```

## Rubyist Convention for getter and setter methods

- ▼ Name those getter and setter methods using the same name as the instance variable they are exposing and setting

```

class GoodDog
  def initialize(name)
    @name = name
  end

  def name # This was renamed from "get_name"
    @name
  end

  def name=(n) # This was renamed from "set_name="
    @name = n
  end

  def speak
    "#{@name} says arf!"
  end
end

sparky = GoodDog.new("Sparky")
puts sparky.speak
puts sparky.name # => "Sparky"
sparky.name = "Spartacus"
puts sparky.name # => "Spartacus"

```

### attr\*

Use the `attr` family of functions to define trivial accessors or mutators.\*

#### ▼ The `attr_accessor` method

The `attr_accessor` method takes a symbol as an argument, which it uses to create the method name for the getter and setter methods. That one line replaced two method definitions. It takes a symbol as a parameter.

```

class GoodDog
  attr_accessor :name

  def initialize(name)
    @name = name
  end

  def speak
    "#{@name} says arf!" # => this could be "#{name} says arf!"
  end # and it would work the same, due to attr_accessor
end # this is the getter instance method name

sparky = GoodDog.new("Sparky")
puts sparky.speak
puts sparky.name # => "Sparky"
sparky.name = "Spartacus"
puts sparky.name # => "Spartacus"

```

#### ▼ `attr_reader` to replace getter only

Use the `attr_reader` method. It works the same way but only allows you to retrieve the instance variable. It takes a symbol as a parameter.

#### ▼ `attr_writer` to replace setter only

If you only want the setter method, you can use the `attr_writer` method. It takes a symbol as a parameter.

#### ▼ Track multiple states by adding them as parameters (symbols only) :

```
attr_accessor :name, :height, :weight
```

`@name`, `@height`, `@weight` are all states here that can be read or reassigned.

#### ▼ Referencing instance variables with `attr*` methods:

With the use of `attr*` methods, you do not have to reference the instance variable directly in a method, you can use the the name of the getter/setter method as seen in the `attr*` method

```
def speak
  "#{name} says arf!" #instead of "#{@name} says arf!"
end
```

## Using `attr_writer` method requirement

To ensure you are not creating a new local variable, you must to call `self.name =` or reference the instance variable directly with `@instance_var_name =`

This tells Ruby that we're calling a **setter** method, not creating a local variable.

#### ▼ Example 1: with `self`

```
def change_info=(n, h, w)
  self.name = n
  self.height = h
  self.weight = w
end
```

#### ▼ Example 2: with `self`

```
class Person
  attr_accessor :first_name, :last_name

  def initialize(full_name)
    parse_full_name(full_name)
  end

  def name
    "#{first_name} #{last_name}".strip
  end

  def name=(full_name)
    parse_full_name(full_name)
  end

  private

  def parse_full_name(full_name)
    parts = full_name.split
    self.first_name = parts.first
    self.last_name = parts.size > 1 ? parts.last : ''
  end
end

bob = Person.new('Robert')
bob.name          # => 'Robert'
bob.first_name    # => 'Robert'
bob.last_name     # => ''
```

```

bob.last_name = 'Smith'
bob.name      # => 'Robert Smith'

```

#### ▼ Example 3: with a **private** `attr_writer`

```

class Dog
  attr_reader :nickname

  def initialize(n)
    @nickname = n
  end

  def change_nickname(n)
    self.nickname = n # you could also write @nickname = n
  end

  def greeting
    "#{nickname.capitalize} says Woof Woof!"
  end

  private
  attr_writer :nickname
end

dog = Dog.new("rex")
dog.change_nickname("barny") # changed nickname to "barny"
puts dog.greeting # Displays: Barny says Woof Woof!

```

Note that the `setter` method for `nickname` is private: it is not available outside of the class and `dog.nickname = "barny"` would raise an error.

You might have noticed that even though the `setter` method for `nickname` is private we are still calling it with `self` prepended on line 9, `self.nickname = n`.

#### ▼ Example 4: demonstrating both options

This code will not work since we're creating a new local variable instead of reassigning the value of the instance variable:

```

class InvoiceEntry
  attr_reader :quantity, :product_name # Notice it's an attr_reader

  def initialize(product_name, number_purchased)
    @quantity = number_purchased
    @product_name = product_name
  end

  def update_quantity(updated_count)
    quantity = updated_count if updated_count >= 0 # Initializes a new local var
  end
end

```

To fix this, there are 2 options:

1. change `attr_reader` to `attr_accessor`, and then use the "setter" method like this: `self.quantity = updated_count if updated_count >= 0`.

##### ▼ Full code

```

class InvoiceEntry
  attr_accessor :quantity, :product_name

  def initialize(product_name, number_purchased)
    @quantity = number_purchased
    @product_name = product_name
  end
end

```

```

end

def update_quantity(updated_count)
  self.quantity = updated_count if updated_count >= 0
end
end

```

2. reference the instance variable directly within the `update_quantity` method, like this `@quantity = updated_count if updated_count >= 0`

▼ Full code

```

class InvoiceEntry
  attr_reader :quantity, :product_name

  def initialize(product_name, number_purchased)
    @quantity = number_purchased
    @product_name = product_name
  end

  def update_quantity(updated_count)
    @quantity = updated_count if updated_count >= 0
  end
end

```

## Design Consideration

When using an `attr_writer` or `attr_accessor`, you are making the methods for the instance variables available in the public interface. If a user knows the name of the instance variable, they can update its value directly instead of going through a method. This reduces data protection.

You could adopt the same with getter methods but it's not required.

## Public, Private, Protected

Ruby gives you three levels of protection at instance methods level, which may be **public**, **private**, or **protected**. Ruby does not apply any access control over instance and class variables.

- A **public** method is a method that is available to anyone who knows either the class name or the object's name. These methods are readily available for the rest of the program to use and comprise the class's interface (that's how other classes and objects will interact with this class and its objects). Methods are public by default except for `initialize`, which is always private.
- ▼ Sometimes you'll have methods that are doing work in the class but don't need to be available to the rest of the program. These methods can be defined as **private**.

Private methods are only accessible from other methods in the class.

We use the `private` method call in our program and anything below it is private (unless another method, like `protected`, is called after it to negate it).



Private methods can never be called with an explicit caller, even when that caller is `self` in Ruby versions prior to 2.7

▼ Example 1: method below `private`

```

class GoodDog
  DOG_YEARS = 7

  attr_accessor :name, :age

```

```

def initialize(n, a)
  self.name = n
  self.age = a
end

private

def human_years
  age * DOG_YEARS
end

end

sparky = GoodDog.new("Sparky", 4)
sparky.human_years

# Gives the following output
NoMethodError: private method `human_years' called for
#<GoodDog:0x007f8f431441f8 @name="Sparky", @age=4>

```

### ▼ Example 2: method above `private`

Private methods are not accessible outside of the class definition at all, and are only accessible from inside the class when called without self.

```

class GoodDog
  DOG_YEARS = 7

  attr_accessor :name, :age

  def initialize(n, a)
    self.name = n
    self.age = a
  end

  def public_disclosure
    "#{self.name} in human years is #{human_years}"
  end

  private

  def human_years
    age * DOG_YEARS
  end

end

sparky = GoodDog.new("Sparky", 4)
p sparky.public_disclosure # "Sparky in human years is 28"

```

### ▼ Example 3: another example

```

class Vehicle
  # code omitted for brevity...
  def age
    "Your #{self.model} is #{years_old} years old."
  end

  private

  def years_old
    Time.now.year - self.year
  end

end

# code omitted for brevity...

puts lumina.age #=> "Your chevy lumina is 17 years old"

```

▼ We can use the `protected` keyword to create **protected** methods. The easiest way to understand protected methods is to follow these two rules:

- from inside the class, `protected` methods are accessible just like `public` methods.
- from outside the class, `protected` methods act just like `private` methods.

#### ▼ Example

```
class Animal
  def a_public_method
    "Will this work? " + self.a_protected_method
  end

  protected

  def a_protected_method
    "Yes, I'm protected!"
  end
end

fido = Animal.new

fido.a_public_method      # => Will this work? Yes, I'm protected!

fido.a_protected_method
# => NoMethodError: protected method `a_protected_method' called for
#<Animal:0x007fb174157110>
```

### Differences between `private` and `protected`

1. Private and protected methods cannot be called from the outside of the class. Access is restricted.
2. Private and protected methods can be called from the inside of the defining class. Access is allowed.
3. Private methods cannot be called with an explicit receiver and protected ones can in Ruby versions prior to `2.7`

#### ▼ Practical Example 1

Protected methods in Ruby come into play when you have two objects of the same type. Sometimes, those two objects need access to each other's state or behavior. If that state or behavior is private, they can't access it. If it's public, than anybody can access it. Protected is the middle ground - it allows access between objects of the same type, but doesn't allow access to anything else.

▼ A common place this comes up is with something like an `==` method:

```
class Something
  def ==(other)
    protected_data == other.protected_data
  end

  protected

  attr_reader :protected_data
end

s1 = Something.new
s2 = Something.new

puts s1 == s2
```

If `protected_data` is private, the `==` method for `s1` can't access `other.protected_data` (where `other` is `s2`).  
If `protected_data` is public, then anything can access it. If it's protected, though, this works.

#### ▼ Practical Example + Explanation 2



```

class Student
  attr_writer :grade

  def better_grade?(other_student)
    grade > other_student.grade
  end

  protected

  attr_reader :grade
end

student1 = Student.new
student1.grade = 76

student2 = Student.new
student2.grade = 83

p student1.better_grade?(student2)

```

In this code we define a `Student` class with 4 methods:

- An `attr_writer` accessor setter method with the symbol `:grade` as an argument. This is an example of Ruby's syntactical sugar and is the equivalent of `def grade=(g); @grade = g; end`. This method allows us to assign a value to the `@grade` instance variable.
- A `#better_grade?` instance method with 1 parameter `other_student`. Within the method we compare the calling object's grade with another instance.
- A `protected` method which only allows instances of the class or a subclass to call the method.
- A `attr_reader` accessor getter method with the symbol `:grade` as an argument. This is an example of Ruby's syntactical sugar and is the equivalent of `def grade; @grade; end`. This method allows us to get the value assigned to the `@grade` instance variable.
- On `line 13` we initialize a local variable `student1` and assign it to the instantiation of a new object of the `Student` class.
- On `line 14` we invoke the `#grade=()` method on the `student1` object and pass in the integer `76` as an argument.

## The `to_s` Method

The `to_s` instance method comes built into every class in Ruby. By default, the `to_s` method returns the name of the object's class and an encoding of the object id.

### ▼ Example with `puts`

```

puts sparky # => #<GoodDog:0x007fe542323320>
#equivalent to
puts sparky.to_s

```

`puts` automatically calls `to_s` on its argument.

### ▼ Note regarding `puts`

`puts` method calls `to_s` for any argument that is not an array. For an array, it writes on separate lines the result of calling `to_s` on each element of the array.

### ▼ Example with string interpolation vs concatenation

In a class where `to_s` is not defined, we will get an enumerator as the return value

```
bob = Person.new("Robert Smith")
puts "The person's name is: #{bob}"
# The person's name is: #<Person:0x007fb873252640>
```

This is because when we use string interpolation (as opposed to string concatenation), Ruby automatically calls the `to_s` instance method on the expression between the `#{}`. Every object in Ruby comes with a `to_s` inherited from the `Object` class. By default, it prints out some gibberish, which represents its place in memory.

If we do not have a `to_s` method that we can use, we must construct the string in some other way. For instance, we can use:

```
puts "The person's name is: " + bob.name      # => The person's name is: Robert Smith
```

or

```
puts "The person's name is: #{bob.name}"      # => The person's name is: Robert Smith
```

## Class: Variables, Methods, Scope, Inheritance

### Class Variables

Class variables are created using `@@` before the name of the variable.

#### ▼ Example of a Class Variable

```
class GoodDog
  @@number_of_dogs = 0

  def initialize
    @@number_of_dogs += 1
  end

  def self.total_number_of_dogs
    @@number_of_dogs
  end
end

puts GoodDog.total_number_of_dogs  # => 0

dog1 = GoodDog.new
dog2 = GoodDog.new

puts GoodDog.total_number_of_dogs  # => 2
```

We have a class variable called `@@number_of_dogs`, which we initialize to 0. Then in our constructor (the `initialize` method), we increment that number by 1. `initialize` gets called every time we instantiate a new object via the `new` method. This also demonstrates that we can access class variables from within an instance method (`initialize` is an instance method). Finally, we just return the value of the class variable in the class method `self.total_number_of_dogs`.

This is an example of using a class variable and a class method to keep track of a class level detail that pertains only to the class, and not to individual objects.

### Class Variable Scope

Class variables start with `@@` and are scoped at the class level. They exhibit two main behaviors:

- All objects share 1 copy of the class variable (including sub-classes)
- This also implies objects can access class variables by way of instance methods.
- Class variables can share state between objects
- Used to track class attributes
- Class methods can access class variables, regardless of where it's initialized

#### ▼ Example: Class Variables

```
class Person
  @@total_people = 0      # initialized at the class level

  def self.total_people
    @@total_people        # accessible from class method
  end

  def initialize
    @@total_people += 1    # mutable from instance method
  end

  def total_people
    @@total_people        # accessible from instance method
  end
end

Person.total_people      # => 0
Person.new
Person.new
Person.total_people      # => 2

bob = Person.new
bob.total_people         # => 3

joe = Person.new
joe.total_people         # => 4

Person.total_people      # => 4
```

From the above, you can see that even when we have two different `Person` objects in `bob` and `joe`, we're effectively accessing and modifying one copy of the `@@total_people` class variable.

We can't do this with instance variables or local variables; only class variables can share state between objects. (We're going to ignore globals.)

## With Inheritance

Class Variables allow sub-classes to override super-class class variables.


The change will affect all other sub-classes of the super-class.

#### ▼ Example: In a sub-classes

```
class Animal
  @@total_animals = 0

  def initialize
    @@total_animals += 1
  end
end

class Dog < Animal
```

 *This is extremely unintuitive behavior, forcing some Rubyists to avoid using class variables altogether.*

#### ▼ Example: What not to do

```
class Vehicle
  @@wheels = 4

  def self.wheels
    @@wheels
  end
end
```

```
def total_animals
  @@total_animals
end

spike = Dog.new
spike.total_animals #1
```

Since this class variable is initialized in the `Animal` class, there is no method to explicitly invoke to initialize it. The class variable is loaded when the class is evaluated by Ruby.

```
Vehicle.wheels # => 4

class Motorcycle < Vehicle
  @@wheels = 2
end

Motorcycle.wheels # => 2
Vehicle.wheels # => 2 Yik

class Car < Vehicle
end

Car.wheels # => 2 Not
```

For this reason, avoid using class variables when working with inheritance. In fact, some Rubyists would go as far as recommending avoiding class variables altogether. The solution is usually to use class instance variables.

## Class Methods

Class methods are where we put functionality that does not pertain to individual objects. Class methods are methods we can call directly on the class itself, without having to instantiate any objects.

▼ When defining a class method, we prepend the method name with the reserved word `self.`

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

def self.what_am_i # Class method definition
  "I'm a GoodDog class!"
end
```

▼ When we call the class method, we use the class name followed by the method name

```
GoodDog.what_am_i # => I'm a GoodDog class!
```

## Class Constants: Scope, Inheritance, Lookup

### Class Constants

There may be certain variables that you never want to change. You can do this by creating what are called **constants**. You define a constant by using an upper case letter at the beginning of the variable name.

While technically constants just need to begin with a capital letter, most Rubyists will make the entire variable uppercase.

▼ Example of a Class Constant

```
class GoodDog
  DOG_YEARS = 7

  attr_accessor :name, :age

  def initialize(n, a)
    self.name = n
    self.age = a * DOG_YEARS
  end
end
```

```

end

sparky = GoodDog.new("Sparky", 4)
puts sparky.age           # => 28

```

We are using the constant `DOG_YEARS` to calculate the age in dog years when we created the object, `sparky`.

`DOG_YEARS` is a variable that will never change for any reason so we use a constant. It is possible to reassign a new value to constants but Ruby will throw a warning.

#### ▼ Example of retrieving a Class constant

```

#!/usr/bin/ruby -w

# define a class
class Box
  BOX_COMPANY = "TATA Inc"
  BOXWEIGHT = 10
  # constructor method
  def initialize(w,h)
    @width, @height = w, h
  end
  # instance method
  def getArea
    @width * @height
  end
end

# create an object
box = Box.new(10, 20)

# call instance methods
a = box.getArea()
puts "Area of the box is : #{a}"
puts Box::BOX_COMPANY
puts "Box weight is: #{Box::BOXWEIGHT}"

```

When the above code is executed, it produces the following result –

```

Area of the box is : 200
TATA Inc
Box weight is: 10

```

## Constant Variable Scope

- Constant variables are usually just called constants, because you're not supposed to re-assign them to a different value.
- If you do re-assign a constant, Ruby will warn you (but won't generate an error).
- Constants have **Lexical scope**, which is defined where a variable is initialized. Inner scope can access outer scope, but not vice versa.

#### ▼ Example: Constants in Class and Instance Methods

```

class Person
  TITLES = ['Mr', 'Mrs', 'Ms', 'Dr']

  attr_reader :name

  def self.titles
    TITLES.join(', ')
  end

  def initialize(n)

```

```

    @name = "#{TITLES.sample} #{n}"
  end
end

Person.titles          # => "Mr, Mrs, Ms, Dr"

bob = Person.new('bob')
bob.name               # => "Ms bob" (output may vary)

```

## With Inheritance

Constants have lexical scope which makes their scope resolution rules very unique compared to other variable types. If Ruby doesn't find the constant using lexical scope, it'll then look at the inheritance hierarchy.

### ▼ Example: Constant in separate class

```

class Dog
  LEGS = 4
end

class Cat
  def legs
    LEGS
  end
end

kitty = Cat.new
kitty.legs          # => NameError: uninitialized constant Cat::LEGS

```

The error occurs here because Ruby is looking for `LEGS` within the `Cat` class. This is expected, since this is the same behavior as class or instance variables (except, referencing an uninitialized instance variable will return `nil`).

### ▼ Example: With namespacing

But unlike class or instance variables, we can actually reach into the `Dog` class and reference the `LEGS` constant. In order to do so, we have to tell Ruby where the `LEGS` constant is using `::`, which is the **namespace resolution operator**.

```

class Dog
  LEGS = 4
end

class Cat
  def legs
    Dog::LEGS          # added the :: namespace resolution operator
  end
end

kitty = Cat.new
kitty.legs             # => 4

```

### ▼ Example: With inheritance in classes

A constant initialized in a super-class is inherited by the sub-class, and can be accessed by both class and instance methods.

```

class Vehicle
  WHEELS = 4
end

class Car < Vehicle
  def self.wheels

```

```

    WHEELS
  end

  def wheels
    WHEELS
  end
end

Car.wheels          # => 4

a_car = Car.new
a_car.wheels        # => 4

```

### ▼ Example: Class constant in super/sub class

```

class Vehicle
  WHEELS = 4

  def self.wheels
    WHEELS
  end

  def wheels
    WHEELS
  end
end

class Car < Vehicle
  WHEELS = 9

  def self.wheels
    WHEELS
  end

  def wheels
    WHEELS
  end
end

a_car = Car.new
p a_car.wheels      # => 9
p Car.wheels        # => 9

other_car = Vehicle.new
p other_car.wheels  # => 4
p Vehicle.wheels    # => 4

puts Car::WHEELS.object_id  # => 9
puts Vehicle::WHEELS.object_id  # => 19

```

### ▼ Example: With mixins

Unlike instance methods or instance variables, constants are not evaluated at runtime, so their lexical scope - or, where they are used in the code - is very important.

```

module Maintenance
  def change_tires
    "Changing #{WHEELS} tires."
  end
end

class Vehicle
  WHEELS = 4
end

class Car < Vehicle
  include Maintenance
end

```

```
a_car = Car.new
a_car.change_tires # => NameError: uninitialized constant Maintenance::WHEELS
```

In this case, the line "Changing #{WHEELS} tires." is in the `Maintenance` module, which is where Ruby will look for the `WHEELS` constant. Even though we call the `change_tires` method from the `a_car` object, Ruby is not able to find the constant.

**To fix the above, we could do the following:**

```
module Maintenance
  def change_tires
    "Changing #{Vehicle::WHEELS} tires." # this fix works
  end
end
```

or

```
module Maintenance
  def change_tires
    "Changing #{Car::WHEELS} tires." # surprisingly, this also works
  end
end
```

The reason `Car::WHEELS` works is because we're telling Ruby to look for `WHEELS` in the `Car` class, which can access `Vehicle::WHEELS` through inheritance.

## self



The **object** the current **method** is attached to.

`self` can refer to different things depending on where it is used.

Thus, we can see that `self` is a way of being explicit about what our program is referencing and what our intentions are as far as behavior. `self` changes depending on the scope it is used in.

## Instance method

▼ `self`, inside of an instance method, references the instance (object) that called the method - the calling object. Therefore, `self.weight=` is the same as `sparky.weight=`, in our example.

```
class GoodDog
  attr_accessor :name, :height, :weight

  def initialize(n, h, w)
    self.name = n
    self.height = h
    self.weight = w
  end

  def change_info(n, h, w)
    self.name = n
    self.height = h
    self.weight = w
  end

  def info
    "#{self.name} weighs #{self.weight} and is #{self.height} tall."
  end
end
```



## class

- ▼ When `self` is **prepended** to a **method definition**, it is defining a class method.

```
class MyAwesomeClass
  def self.this_is_a_class_method
  end
end
```

A method definition prefixed with `self` is the same as defining the method on the class.

That is, `def self.a_method` is equivalent to `def GoodDog.a_method`.

`self`, outside of an instance method, references the class and can be used to define class methods. Therefore, `def self.name=(n)` is the same as `def GoodDog.name=(n)`, in our example.

## Using `self`

- ▼ Prefixing `self` is not restricted to just the accessor methods; you can use it with any instance method. For example, the `info` method is not a method given to us by `attr_accessor`, but we can still call it using `self.info`:

```
class GoodDog
  # ... rest of code omitted for brevity
  def some_method
    self.info
  end
end
```

# Module



A class can only sub-class from one parent, but it can mix in as many modules as it likes.

## Main purpose: Supporting multiple inheritance

A module is a collection of behaviors that is usable in other classes via mixins. A module is *mixed in* to a class using the `include` method invocation.

- ▼ **Example: the `speak` method in many classes**

```
module Speak
  def speak(sound)
    puts sound
  end
end

class GoodDog
  include Speak
end

class HumanBeing
  include Speak
end

sparky = GoodDog.new
sparky.speak("Arf!")      # => Arf!
bob = HumanBeing.new
bob.speak("Hello!")      # => Hello!
```

Mixing in the module `Speak` to the classes `GoodDog` and `HumanBeing` allows us to invoke the method `speak` on the objects of both the classes.

## Namespacing

Namespacing is a way of bundling logically related objects together. Modules serve as a convenient tool for this. This allows classes or modules with conflicting names to co-exist while avoiding collisions. Think of this as storing different files with the same names under separate directories in your file system.

### ▼ Example 1: Namespacing

```
module Perimeter
  class Array
    def initialize
      @size = 400
    end
  end
end

our_array = Perimeter::Array.new
ruby_array = Array.new

p our_array.class
p ruby_array.class
```

### ▼ Example 2: Advantages & Invocation

In this context, namespacing means **organizing similar classes under a module**. In other words, we'll use modules to group related classes. It becomes easy for us to recognize related classes in our code.

The second advantage is it **reduces the likelihood of our classes colliding** with other similarly named classes in our codebase.

```
module Mammal
  class Dog
    def speak(sound)
      p "#{sound}"
    end
  end

  class Cat
    def say_name(name)
      p "#{name}"
    end
  end
end
```

We **call classes in a module** by appending the class name to the module name with two colons(`::`)

```
buddy = Mammal::Dog.new
kitty = Mammal::Cat.new
buddy.speak('Arf!')      # => "Arf!"
kitty.say_name('kitty')  # => "kitty"
```

### ▼ Example 3: Container for methods/module methods

Using modules as a container for methods, called **module methods**. This involves using modules to house other methods. This is very useful for methods that seem out of place within your code.

```
module Mammal
  ...
```

```

def self.some_out_of_place_method(num)
  num ** 2
end
end

```

Defining methods this way within a module means we can call them directly from the module:

```
value = Mammal.some_out_of_place_method(4)
```

We can also call such methods by doing:

```
value = Mammal::some_out_of_place_method(4)
```

although the former is the preferred way.

## Mixing in Modules

Using modules allows us to DRY up our code.

### ▼ module syntax example

```

module Towable
  def can_tow?(pounds)
    pounds < 2000 ? true : false
  end
end

class Vehicle
  @@number_of_vehicles = 0

  def self.number_of_vehicles
    puts "This program has created #{@@number_of_vehicles} vehicles"
  end

  def initialize
    @@number_of_vehicles += 1
  end

  def self.gas_mileage(gallons, miles)
    puts "#{miles / gallons} miles per gallon of gas"
  end
end

class MyCar < Vehicle
  NUMBER_OF_DOORS = 4
  #code omitted for brevity...
end

class MyTruck < Vehicle
  include Towable

  NUMBER_OF_DOORS = 2
end

```

### ▼ Grouping behaviors + convention

Grouping behaviors that are common but shouldn't be part of a class do well in a `module`

```

module Swimmable
  def swim
    "I'm swimming!"
  end
end

```

```

class Animal; end

class Fish < Animal
  include Swimmable      # mixing in Swimmable module
end

class Mammal < Animal
end

class Cat < Mammal
end

class Dog < Mammal
  include Swimmable      # mixing in Swimmable module
end

sparky = Dog.new
neemo  = Fish.new
paws   = Cat.new

sparky.swim      # => I'm swimming!
neemo.swim       # => I'm swimming!
paws.Swim        # => NoMethodError: undefined method `swim' for #<Cat:0x007fc453152308>

```



Note: A common naming convention for Ruby is to use the "able" suffix on whatever verb describes the behavior that the module is modeling. You can see this convention with our `Swimmable` module. Likewise, we could name a module that describes "walking" as `Walkable`. Not all modules are named in this manner, however, it is quite common.

- See polymorphism for relationship

## Tip

Avoid prepending a method definition with `self` as this will always refer to the module rather than the class the module is mixed in.

## Method Lookup Path

Method lookup path is the chain of places Ruby will go to find the method we are trying to invoke. It starts with the current class, the most recent included module, and goes up all the way to the `BasicObject` class.

This goes for every method invocation, including methods invoked within the body of another method. Ruby always starts at the object's class and then goes up the inheritance tree.

### Method Lookup Path

The method lookup path is the order in which classes are inspected when you call a method.

#### ▼ Example 1: 3 modules and 1 class

```

module Walkable
  def walk
    "I'm walking."
  end
end

module Swimmable
  def swim
    "I'm swimming."
  end
end

module Climbable

```

```

def climb
  "I'm climbing."
end

class Animal
  include Walkable

  def speak
    "I'm an animal, and I speak!"
  end
end

puts "---Animal method lookup---"
puts Animal.ancestors

# ---Animal method lookup---
# Animal
# Walkable
# Object
# Kernel
# BasicObject

```

### ▼ Example 2: Another class added

```

class GoodDog < Animal
  include Swimmable
  include Climbable
end

puts "---GoodDog method lookup---"
puts GoodDog.ancestors

# ---GoodDog method lookup---
# GoodDog
# Climbable
# Swimmable
# Animal
# Walkable
# Object
# Kernel
# BasicObject

```

First, this tells us that the order in which we include modules is important.

Ruby looks at the **last** module we included **first**. This means that in the rare occurrence that the modules we mix in contain a method with the same name, the last module included will be consulted first. The second interesting thing is that the module included in the superclass made it on to the method lookup path.

That means that all `GoodDog` objects will have access to not only `Animal` methods, but also methods defined in the `Walkable` module, as well as all other modules mixed in to any of its superclasses.

**We can use `ancestors` methods on any class to find out the method lookup chain**

```

module Speak
  def speak(sound)
    puts "#{sound}"
  end
end

class GoodDog
  include Speak
end

class HumanBeing
  include Speak
end

```

```
puts "---GoodDog ancestors---"
puts GoodDog.ancestors
puts ''
puts "---HumanBeing ancestors---"
puts HumanBeing.ancestors
```

And the output would look like this:

```
---GoodDog ancestors---
GoodDog
Speak
Object
Kernel
BasicObject

---HumanBeing ancestors---
HumanBeing
Speak
Object
Kernel
BasicObject
```

The `Speak` module is placed right in between our custom classes (i.e., `GoodDog` and `HumanBeing`) and the `Object` class that comes with Ruby.

This means that since the `speak` method is not defined in the `GoodDog` class, the next place it looks is the `Speak` module. This continues in an ordered, linear fashion, until the method is either found, or there are no more places to look.

## Fake Operator

### Methods Vs Operators

Overriding a method is possible, Overriding an operator is not.

#### ▼ Syntactic Sugar

It only *looks* like an operator because Ruby gives us special **syntactical sugar** when invoking that method. Instead of calling the method normally (eg. `str1.==(str2)`), we can call it with a special syntax that reads more naturally (eg. `str1 == str2`).

Method	Operator	Description
yes	<code>[]</code> , <code>[]=</code>	Collection element getter and setter.
yes	<code>**</code>	Exponential operator
yes	<code>!</code> , <code>~</code> , <code>+</code> , <code>-</code>	Not, complement, unary plus and minus (method names for the last two are <code>+</code> @ and <code>-</code> @)
yes	<code>*</code> , <code>/</code> , <code>%</code>	Multiply, divide, and modulo
yes	<code>+</code> , <code>-</code>	Plus, minus
yes	<code>&gt;&gt;</code> , <code>&lt;&lt;</code>	Right and left shift
yes	<code>&amp;</code>	Bitwise "and"
yes	<code>^</code> , <code> </code>	Bitwise exclusive "or" and regular "or" (inclusive "or")
yes	<code>&lt;=</code> , <code>&lt;</code> , <code>&gt;</code> , <code>&gt;=</code>	Less than/equal to, less than, greater than, greater than/equal to
yes	<code>&lt;=&gt;</code> , <code>==</code> , <code>===</code> , <code>!=</code> , <code>=~</code> , <code>!~</code>	Equality and pattern matching ( <code>!=</code> and <code>!~</code> cannot be directly defined)
no	<code>&amp;&amp;</code>	Logical "and"
no	<code>  </code>	Logical "or"
no	<code>..</code> , <code>...</code>	Inclusive range, exclusive range
no	<code>? :</code>	Ternary if-then-else
no	<code>=</code> , <code>%=</code> , <code>/=</code> , <code>-=</code> , <code>+=</code> , <code> =</code> , <code>&amp;=</code> , <code>&gt;&gt;=</code> , <code>&lt;&lt;=</code> , <code>*=</code> , <code>&amp;&amp;=</code> , <code>  =</code> , <code>**=</code> , <code>{</code>	Assignment (and shortcuts) and block delimiter

## Fake Operator: Comparison Method `#>`

Defining a `>` method in our class can be done by relying on another implementation:

- ▼ Pushing the comparison functionality to the `Integer#>` method:

Assume `age` in this case in an Integer.

```
def >(other_person)
  age > other_person.age
end
```

Defining a `>` doesn't provide a `<`, it must be defined separately

- ▼ Full Code Example & Syntactical Sugar Comparison

```
class Person
  attr_accessor :name, :age

  def initialize(name, age)
    @name = name
    @age = age
  end

  def >(other_person)
    age > other_person.age
  end
end

bob = Person.new("Bob", 49)
kim = Person.new("Kim", 33)

puts "bob is older than kim" if bob > kim # syntactic sugar notation
```

```
puts "bob is older" if bob > kim      # syntactic sugar notation
puts "bob is older" if bob.>(kim)     # explicit notation
```

## Fake Operator: Shift Method #<<

For example, using the << method fits well when working with a class that represents a **collection**.

### ▼ Example Implementation of #<< in a custom class

```
class Team
  attr_accessor :name, :members

  def initialize(name)
    @name = name
    @members = []
  end

  def <<(person)
    members.push person
  end
end

cowboys = Team.new("Dallas Cowboys")
emmitt = Person.new("Emmitt Smith", 46)

cowboys << emmitt

cowboys.members      # => [#<Person:0x007fe08c209530>]
```

Implementing a Team#<< allows for a very clean interface. It makes sense to use it with the @members instance variable since it stores the elements in an array.

### ▼ Example refactoring of previous code

```
class Team
  attr_accessor :name, :members

  def initialize(name)
    @name = name
    @members = []
  end

  def <<(person)
    members << person    # members returns an array, we can rely on Array#<<
                          # instead of Array#push
  end
end

cowboys = Team.new("Dallas Cowboys")
emmitt = Person.new("Emmitt Smith", 46)

cowboys << emmitt

cowboys.members      # => [#<Person:0x007fe08c209530>]
```

## Fake Operator: The plus method

### ▼ Sugared Notion Vs Explicit Notation

```
1 + 1      # => 2
1.+(1)     # => 2
```

### ▼ There's a pattern within Ruby as to how this method is used based on the object it's called on:



- `Integer#+` : **increments** the value by value of the argument, returning a new integer
- `String#+` : **concatenates** with argument, returning a new string
- `Array#+` : **concatenates** with argument, returning a new array
- `Date#+` : **increments** the date in days by value of the argument, returning a new date

You should ideally implement `#+` in a way that either increments or concatenates objects.

Pay attention to the **object** being **returned**. If you rely on one of Ruby's methods, you may end up with an object that is not the custom class you implemented. In these cases, you may need to initialize a new object within your fake operator's logic and have it returned.

▼ Example implementation of `#+` within a custom class / return value custom object

```
class Person
  attr_accessor :name, :age

  def initialize(name, age)
    @name = name
    @age = age
  end

  def >(other_person)
    age > other_person.age
  end
end

class Team
  attr_accessor :name, :members

  def initialize(name)
    @name = name
    @members = []
  end

  def <<(person)
    members.push person
  end

  def +(other_team)
    temp_team = Team.new("Temporary Team")
    temp_team.members = members + other_team.members
    temp_team
  end
end

cowboys = Team.new("Dallas Cowboys")
cowboys << Person.new("Troy Aikman", 48)
cowboys << Person.new("Emmitt Smith", 46)
cowboys << Person.new("Michael Irvin", 49)

niners = Team.new("San Francisco 49ers")
niners << Person.new("Joe Montana", 59)
niners << Person.new("Jerry Rice", 52)
niners << Person.new("Deion Sanders", 47)

dream_team = cowboys + niners

p dream_team #It's a Team object vs a simple Array
```

## Fake Operators: Element setter and getter methods

▼ Array getter notations ( `Array#[]` )

```
my_array = %w(first second third fourth) # ["first", "second", "third", "fourth"]

# element reference
```

```
my_array[2]          # => "third"
my_array.[](2)       # => "third"
```

Both these notation are the same method, the first is with syntactical sugar, the other without.

#### ▼ Array setter notations ( `Array#[]=` )

```
my_array = %w(first second third fourth)  # ["first", "second", "third", "fourth"]

# element assignment
my_array[4] = "fifth"
puts my_array.inspect                     # => ["first", "second", "third", "fourth", "fifth"]

my_array.[](5, "sixth")
puts my_array.inspect                     # => ["first", "second", "third", "fourth", "fifth", "sixth"]
```

To implement them, based on a collection in our class, we need to define them using the explicit notation.

#### ▼ Example of custom defined getter/setter

```
class Team
  # ... rest of code omitted for brevity
  # @members is an array

  def [](idx)
    members[idx]
  end

  def []=(idx, obj)
    members[idx] = obj
  end
end
```

## General Design Consideration

When implementing fake operators, choose some functionality that makes sense when used with the special operator-like syntax.

Follow the general usage of the standard libraries.

Pay attention to the **object** being **returned**. If you rely on one of Ruby's methods, you may end up with an object that is not the custom class you implemented.

## Equivalence

### Comparison in Ruby

Two main elements come into play when we talk about comparison/equality:

1. Object Value
2. Object ID
3. \* Same Value and Class, Different Object (less used)

Only objects can be compared. The variable is but a label for the object.

Two variables can reference different **object** but have the same **value**.

Two variables can reference the same object, return the same Object ID and therefore, the same object values.

Two variables can reference different objects that have the same value and are from the same **class**.

**We must know exactly what we are referring to when we talk about equality.**

Certain objects in Ruby are immutable and only have 1 instance, therefore, there cannot be duplicate copies in the Class of the same value. Ex: Integers, Symbols, Booleans, `nil`.

## Ruby's Equality Instance Methods

`==`

Defined in `BasicObject#==`. As per the documentation:

*At the Object level, `==` returns `true` only if obj and other are the same object.*

You will want to create a custom implementation of `==` in your classes and decide what **values** you want to compare.

When you define `==` in your class, you will also get a `!=` for free.

In practice, you will use it's syntactic sugar version (`str1 == str2`) vs explicit (`str1.==(str2)`). It is a method invocation after all.

For most objects, the `==` operator compares the **values** of the **objects**, and is frequently used this way.

`===`

When used by a `case` statement, it will check if the value is part of a group.

Check the class' custom implementation to know the logic.

### ▼ Example 1: `case` statement

```
num = 25

case num
when 1..50
  puts "small number"
when 51..100
  puts "large number"
else
  puts "not in range"
end

# Can be seen as

num = 25

if (1..50) === num      # Relies on Range#===
  puts "small number"
elsif (51..100) === num
  puts "large number"
else
  puts "not in range"
end
```

When `===` compares two objects, such as `(1..50)` and `25`, it's essentially asking "if `(1..50)` is a group, would `25` belong in that group?" In this case, the answer is "yes".

### ▼ Example 2: String and Integer

```
String === "hello" # => true
String === 15      # => false
```

On line 1, `true` is returned because `"hello"` is an instance of `String`, even though `"hello"` doesn't equal `String`. Similarly, `false` is returned on line 2 because `15` is an integer, which doesn't equal `String` and isn't an instance of the `String` class.

`eq1?`

The `eq1?` method determines if two objects contain the **same value** and if they're of the **same class**.

This method is used most often by `Hash` to determine equality among its members. It's not used very often.

`equal?`

Used to check if the two variables are pointing to the **same object in memory**.

Defined in `BasicObject`, can be overridden but shouldn't.

`object_id`

`object_id` is a method we can call on a variable and it will return an integer representation of the object it's referencing location in memory.

## Design Considerations

Rely on the methods Ruby has included in the Core API and it's usage conventions

If a method should be overridden in your custom classes, understand the return values of the implementation

## Collaborator Objects

### Objects, state and instance variables

Objects that are stored as state within another object are also called "collaborator objects". We call such objects collaborators because they work in conjunction (or in collaboration) with the class they are associated with.

### Term Collaborator Object

When we work with collaborator objects, they are usually **custom objects** (e.g. defined by the programmer and not inherited from the Ruby core library)

- ▼ We've set `bob`'s `@pet` instance variable to `bud`, which is a `Bulldog` object. This means that when we call `bob.pet`, it is returning a `Bulldog` object. **`bob` has a collaborator object stored in the `@pet` instance variable.**

```
class Person
  attr_accessor :name, :pet

  def initialize(name)
    @name = name
  end
end

bob = Person.new("Robert")
bud = Bulldog.new          # assume Bulldog class from previous assignment

bob.pet = bud
```

- ▼ Methods on a collaborator object

This means that when we call `bob.pet`, it is returning a `Bulldog` object.

```
bob.pet          # => #<Bulldog:0x007fd8399eb920>
bob.pet.class    # => Bulldog
```

Because `bob.pet` returns a `Bulldog` object, we can then chain any `Bulldog` methods at the end as well:

```
bob.pet.speak    # => "bark!"
bob.pet.fetch    # => "fetching!"
```

When we need that `Bulldog` object to perform some action (i.e. we want to access some behavior of `@pet`), then we can go through `bob` and call the method on the object stored in `@pet`, such as `speak` or `fetch`.

Collaborator objects aren't strictly custom objects. Even the string object stored in `@name` within `bob` in the code above is technically a collaborator object.

## Design Considerations

*Collaborator objects play an important role in object oriented design, since they also represent the connections between various actors in your program. When working on an object oriented program be sure to consider what collaborators your classes will have and if those associations make sense, both from a technical standpoint and in terms of modeling the problem your program aims to solve.*

### ▼ Example: Many collaborator objects

```
class Person
  attr_accessor :name, :pets

  def initialize(name)
    @name = name
    @pets = []      #an array object to collect pets
  end
end

bob = Person.new("Robert")

kitty = Cat.new      # collaborator object
bud = Bulldog.new    # collaborator object

bob.pets << kitty
bob.pets << bud

bob.pets # => [#<Cat:0x007fd839999620>, #<Bulldog:0x007fd839994ff8>]
```