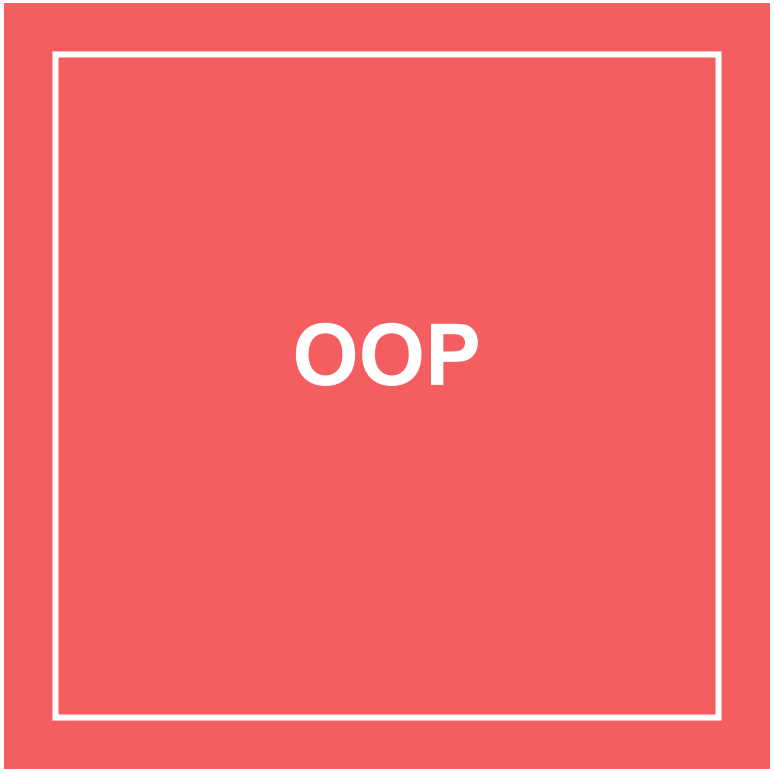

Outline

1. Why Object Oriented Programming?
2. Objects
3. Classes
4. Abstraction
5. Inheritance
6. Encapsulation
7. Polymorphism

1. Why Object Oriented Programming?

Object Oriented Programming, often referred to as **OOP**, is a programming paradigm that was created to deal with the growing complexity of large software systems. Programmers found out very early on that as applications grew in complexity and size, they became very difficult to maintain. One small change at any point in the program would trigger a ripple effect of errors due to dependencies throughout the entire program.

2. Objects

Throughout the Ruby community you'll often hear the phrase, "In Ruby, everything is an object!". We've avoided this reality so far because objects are a more advanced topic and it's necessary to get a handle on basic Ruby syntax before you start thinking about objects.

```
irb :001 > "hello".class
=> String
irb :002 > "world".class
=> String
```

3. Classes - Class Definition

- Ruby defines the attributes and behaviors of its objects in classes. You can think of classes as basic outlines of what an object should be made of and what it should be able to do.
- We replace the def with class and use the CamelCase naming convention to create the name. We then use the reserved word end to finish the definition. Ruby file names should be in snake_case, and reflect the class name. So in the below example, the file name is good_dog.rb and the class name is GoodDog.

```
class GoodDog
end

sparky = GoodDog.new
```

3. Classes - Initialize Method

The initialize method is a standard Ruby class method and works almost same way as constructor works in other object oriented programming languages. The initialize method is useful when you want to initialize some class variables at the time of object creation. This method may take a list of parameters and like any other ruby method it would be preceded by def keyword as shown below

```
class Box
  def initialize w,h
    @width, @height = w, h
  end
end
```

3. Classes - Instance Variable

The instance variables are kind of class attributes and they become properties of objects once objects are created using the class. Every object's attributes are assigned individually and share no value with other objects. They are accessed using the @ operator within the class but to access them outside of the class we use public methods, which are called accessor methods. If we take the above defined class Box then **@width** and **@height** are instance variables for the class Box.

```
class Box
  def initialize w, h
    # assign instance variables
    @width, @height = w, h
  end
end
```

3. Classes - Accessor & Setter

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

```ruby
#!/usr/bin/ruby -w

# define a class
class Box
   # constructor method
   def initialize w, h
      @width, @height = w, h
   end

   # accessor methods
   def printWidth
      @width
   end

   def printHeight
      @height
   end
end
```

```ruby
# create an object
box = Box.new 10, 20

# use accessor methods
x = box.printWidth
y = box.printHeight

puts "Width of the box is : #{x}"
puts "Height of the box is : #{y}"
```

3. Classes - Accessor & Setter (2)

Similar to accessor methods, which are used to access the value of the variables, Ruby provides a way to set the values of those variables from outside of the class using setter methods, which are defined as below

```ruby
#!/usr/bin/ruby -w

# define a class
class Box
  # constructor method
  def initialize w,h
    @width, @height = w, h
  end

  # accessor methods
  def getWidth
    @width
  end

  def getHeight
    @height
  end

  # setter methods
  def setWidth= value
    @width = value
  end
  def setHeight= value
    @height = value
  end
end

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

# use setter methods
box.setWidth = 30
box.setHeight = 50

# use accessor methods
x = box.getWidth
y = box.getHeight

puts "Width of the box is : #{x}"
puts "Height of the box is : #{y}"
```

3. Classes - Instance Method

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.

```ruby
#!/usr/bin/ruby -w

# define a class
class Box
  # constructor method
  def initialize w, h
    @width, @height = w, h
  end

  # instance method
  def getArea
    @width * @height
  end
end
```

```ruby
# create an object
box = Box.new 10, 20

# call instance methods
a = box.getArea
puts "Area of the box is : #{a}"
```

3. Classes - Class Method

A class method is defined using def self.methodname(), which ends with end delimiter and would be called using the class name as classname.methodname as shown in the following example

```ruby
#!/usr/bin/ruby -w

class Box
  # Initialize our class variables
  @@count = 0
  def initialize w, h
    # assign instance avriables
    @width, @height = w, h

    @@count += 1
  end

  def self.printCount
    puts "Box count is : #@@count"
  end
end
```

```ruby
# create two object
box1 = Box.new 10, 20
box2 = Box.new 30, 100

# call class method to print box count
Box.printCount
```

4. Abstraction

In object design we need to define the characteristics of each object and design how they interact with each other.

=> Objects finish work internally, report or change its state and communicate with other objects without knowing how the object proceeds..

5. Inheritance

```
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!
```

5. Inheritance (2)

```ruby
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
```

```ruby
sparky = GoodDog.new "Sparky"
paws = Cat.new

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

6. Encapsulation

- 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).
- 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. How do we define private methods? We use the reserved word private in our program and anything below it is private (unless another reserved word is placed after it to negate it).

6. Encapsulation (2)

```ruby
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
```

6. Encapsulation (3)

Public and private methods are most common, but in some less common situations, we'll want an in-between approach. We can use the protected keyword to create protected methods.

The easiest way to understand protected methods is to follow these two rules:

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

6. Encapsulation (4)

```
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
```

6. Encapsulation (5)

```
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>
```

7. Polymorphism

Though you can add new functionality in a derived class, but sometimes you would like to change the behavior of already defined method in a parent class. You can do so simply by keeping the method name same and overriding the functionality of the method as shown below in the example

```
#!/usr/bin/ruby -w

# define a class
class Box
  # constructor method
  def initialize w,h
    @width, @height = w, h
  End

  # instance method
  def getArea
    @width * @height
  end
end
```

```
# define a subclass
class BigBox < Box
  # change existing getArea method as follows
  def getArea
    @area = @width * @height
    puts "Big box area is : #@area"
  end
end

# create an object
box = BigBox.new 10, 20

# print the area using overriden method.
box.getArea
```

References

1. http://ruby-doc.org/
2. https://launchschool.com/books/oo_ruby/read/inheritance#classinheritance
3. https://www.tutorialspoint.com/ruby/ruby_object_oriented.htm
4. http://ruby.bastardsbook.com/chapters/oops/

Thank you for listening!