Classes and Objects

Objects are data structures that you can call methods on. In Ruby everything is an object including classes.

Difference between Class and Instance Methods

```
class Invoice
  # Class method
 def self.print_out
   "Printed out invoice"
 end

 # Instance method
 def convert_to_pdf
   "Converted to PDF"
 end
end

 # How to call a class with a class method
Invoice.print_out
Output = "Printed out invoice"

 # How to call a class with an instance method
# You have to create a new instance of that class
n = Invoice.new

n.convert_to_pdf
Output = "Converted to PDF"
```

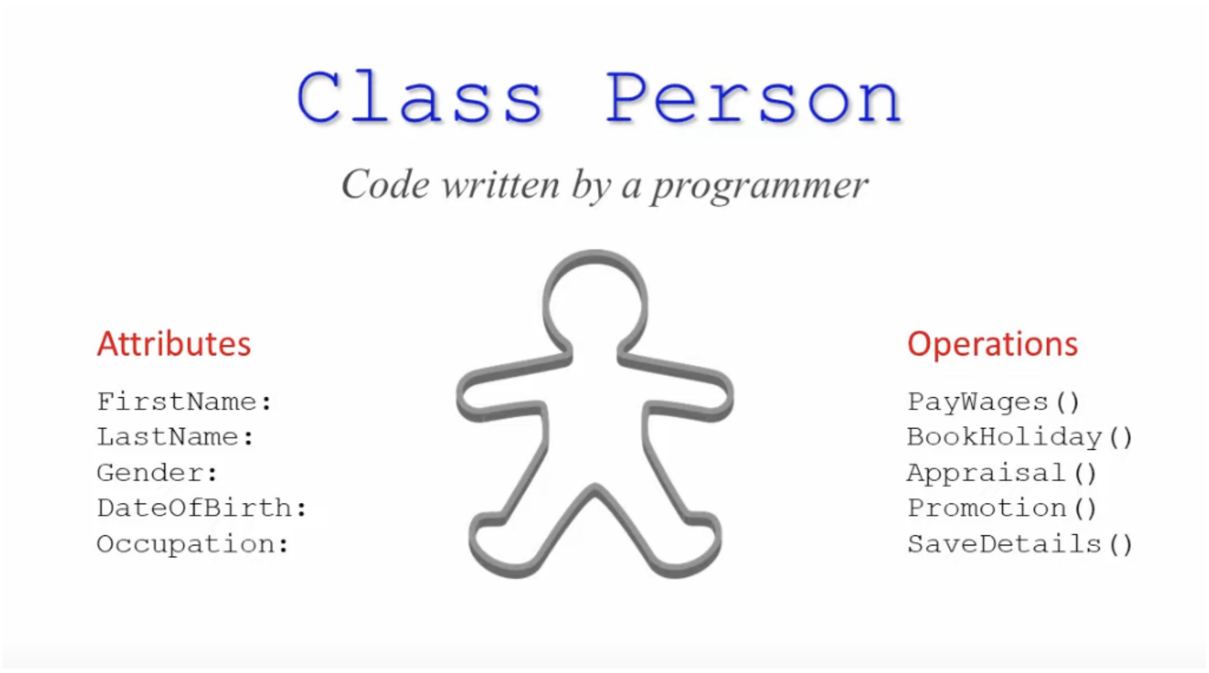
Class methods can be called from anywhere in the program by themselves without having to create an instance variable

Whereas with instance methods, you have to create an instance of the class to be able to use that method

States and behaviours

States - track attributes for individual objects.

Behaviours - are what objects are capable of doing.



💡 Attributes are also called Properties

Operations also called Behaviours or Methods

The property values assigned to an object are collectively known as the **state** of the object.

Initialising a New object

```
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.

Yes, 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.

Instance variables

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

The `@name` variable looks different because it has the `@` symbol in front of it. This is called an **instance variable**. It is a variable that exists as long as the object instance exists and it is one of the ways we tie data to objects.

In the above example, our `initialize` method takes a parameter called `name`. You can pass arguments into the `initialize` method through the `new` method. Let's create an object using the `GoodDog` class from above:

```
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.

Instance Methods

Fairly straightforward, methods that are available on classes are called *class methods*, and methods that are available on instances are called *instance methods*.

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

# Instance Method
  def speak
    "Arf!"
  end
end
########################

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

Getter and Setter methods: attr_accessor

```
class GoodDog
  attr_accessor :name

  def initialize(name)
    @name = name
  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"
```

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.

The `attr_accessor` method creates an instance variable and a method to call them at the same time

But what if we only want the `getter` method without the `setter` method? Then we would want to use the `attr_reader` method. It works the same way but only allows you to retrieve the instance variable. And if you only want the setter method, you can use the `attr_writer` method. All of the `attr_*` methods take a `Symbol` as parameters; if there are more states you're tracking, you can use this syntax:

```
class Human
   # attribute accessors
   attr_accessor :name, :gender, :age


# class variable
@@all = []


   def initialize(name, gender, age)
      @name = name
      @gender = gender
      @age = age
      @@all = self
   end
end


  # instance methods
def print_human
  puts "Name: #{@name}"
  puts "gender: #{@gender}"
  puts "age: #{@age}"
end
```

```
puts [human1.name](http://human1.name)
prints John
```

```
human1.name ="James"
prints James

human1.print_human
prints Name: James
       gender: male
       age: 35
```

```
# Creating several instance variables

human1 = Human.new("John", "male", 35)
human2 = Human.new("Amy", "female", 25)
human3 = Human.new("Stewie", "male", 45)

puts human1.name
prints John
```

Class Methods

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.`, like this:

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

# Class method definition

def self.what_am_i
  "I'm a GoodDog class!"
end
```

Calling the class method:

Then when we call the class method, we use the class name `GoodDog` followed by the method name, without even having to instantiate any objects, like this:

```
GoodDog.what_am_i

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

Class variables

Class variables are created using @@ and must be initialized before they can be used in method definitions.

```
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.

Remember that `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.

Constants

When creating classes there may also 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 letters.

```
class GoodDog
  DOG_YEARS = 7

  attr_accessor :name, :age

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

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

Here we used the constant `DOG_YEARS` to calculate the age in dog years when we created the object, `sparky`.

Note that we used the setter methods in the `initialize` method to initialize the `@name` and `@age` instance variables given to us by the `attr_accessor` method. We then used the `age` getter method to retrieve the value from the object.