***"UNIVERSIDAD DE LAS FUERZAS ARMADAS"***

***ESPE***

***SOFTWARE ENGINEERING***

***OBJECT-ORIENTED PROGRAMMING***

**TEAM 6 : EncryptMPP**


***Moreira Vinueza Erick Patricio***

***Pabon Gonzalez Elkin Andres***

***Ponce Arguello Diego Armando***


***TOPIC:***

---

***Other Programming Language - Ruby***

***INSTRUCTOR:***

*PHD. EDISON LASCANO*

***NRC:***

*4680*

***DATE:***

*July 7, 2022*

***SEDE MATRIZ SANGOLQUÍ***

***QUITO-ECUADOR***

***2022***

## 1.What is Ruby?

Ruby is an **interpreter**, high-level, general-purpose programming language which supports multiple programming paradigms.

In this case we are going to use it with an object-oriented paradigm.

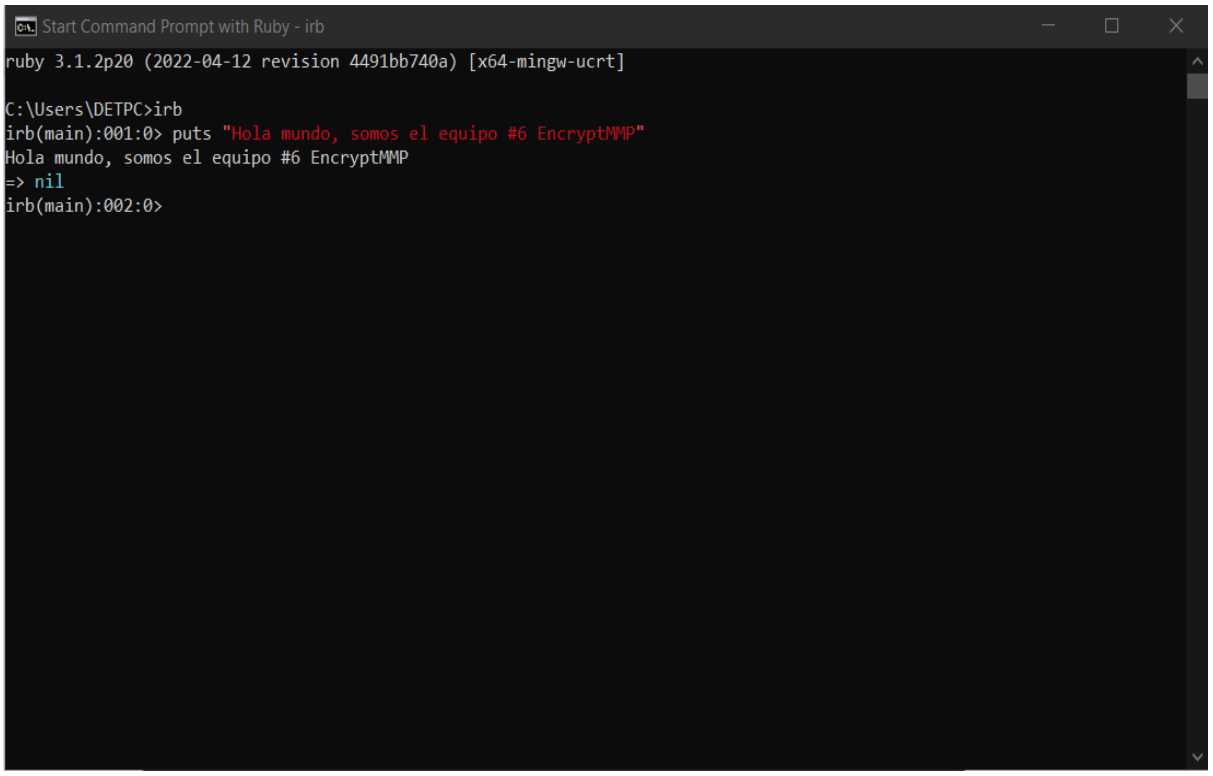
Developer → Ruby was created in the mid-1990s by Yukihiro "Matz" Matsumoto in Japan.

It has there qualities:

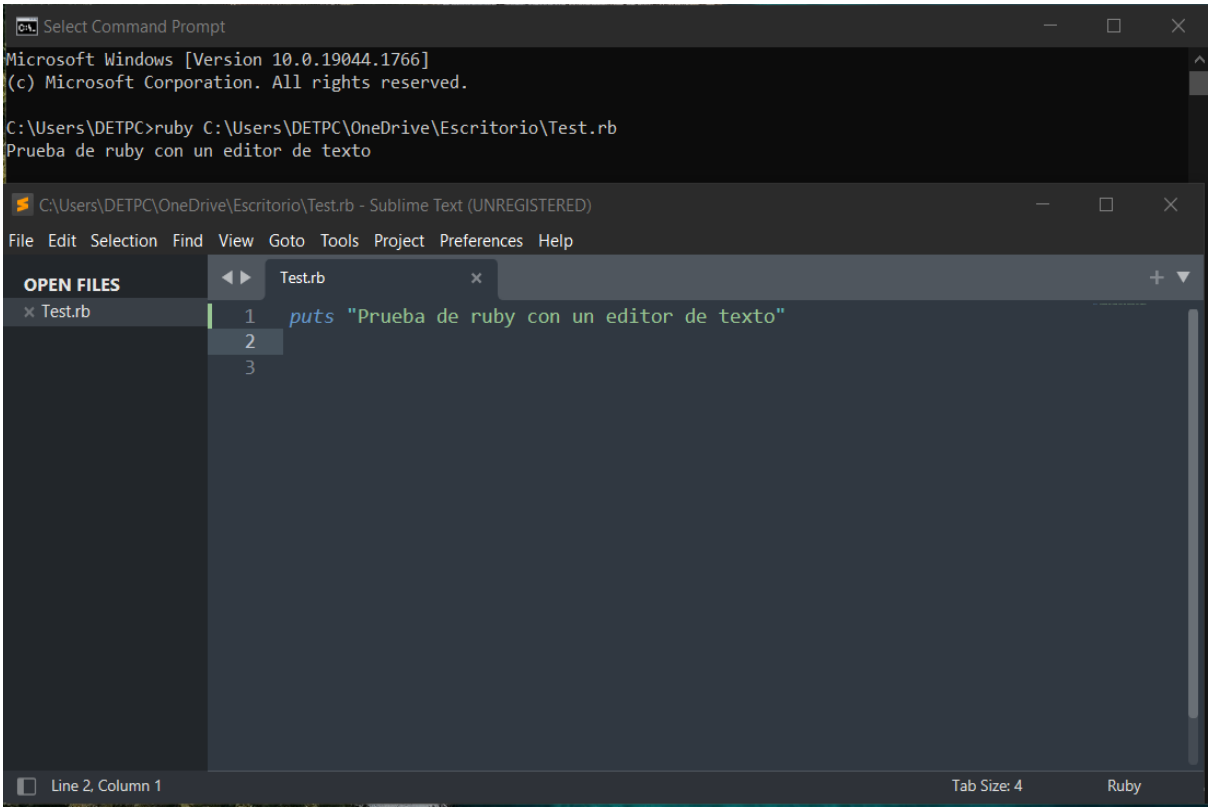
- Terse. Short, but still easy to understand.
- Dynamic. Easy to change, anytime and anywhere.
- Duck typing. If you think you understand it, you probably understand it.

Duck typing → Duck typing is a concept related to dynamic typing, where the type or the class of an object is less important than the methods it defines. When you use duck typing, you do not check types at all. Instead, you check for the presence of a given method or attribute.

## 2.Installation of development environment



As we can see the instruction "Hola mundo, somos el equipo #6 EncryptMPP" **don't need ";"** to print the string.
Also nil **means that the program has finished.**

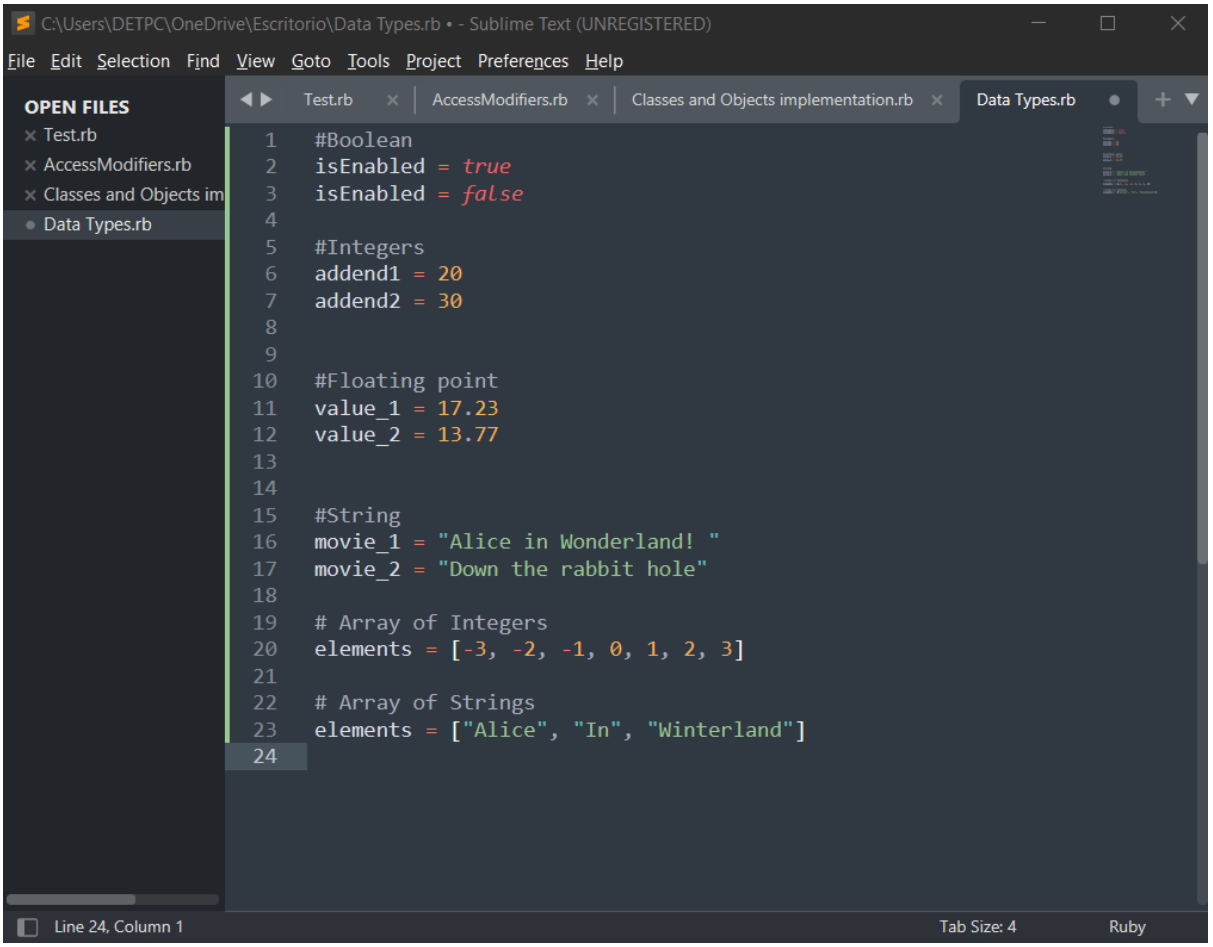



Obviously we can program in a text editor for code, for example Sublime text but we have to save the file with the format .rb, then we can run the program in the command prompt as we can see in the above example.

### 3. Data Types

Common Data types in Ruby are:

- Boolean
- Integer
- Floating point
- String
- Array




**4. Classes and Objects implementation**

A **Class** is a combination of data (attributes) and functions (methods).

The data and functions of a class can be accessed after creating an instance of a class.

Instance of a class is known as an **Object**.

Let us see an example of a Class in Ruby:

```ruby
class Employee
    def initialize
        @employeeId = 0
        @employeeName = ""
    end

    def setEmployeeId(empId)
        @employeeId = empId
    end

    def setEmployeeName(empName)
        @employeeName = empName
    end

    def getEmployeeId
        return @employeeId
    end

    def getEmployeeName
        return @employeeName
    end
end

employee1 = Employee.new
employee1.setEmployeeId(1)
employee1.setEmployeeName("Erick")
puts "El ID del trabajador es #{employee1.getEmployeeId()}"
puts "El nombre del trabajador es #{employee1.getEmployeeName()}
```

```
C:\Users\DETPC>ruby "C:\Users\DETPC\OneDrive\Escritorio\Classes and Objects implementation.rb"
El ID del trabajador es 1
El nombre del trabajador es Erick

C:\Users\DETPC>
```

In this example the first step was to create the class “Employee”, and as we can see in the line #24 We can create objects of Employee class as we did up there.
The getters and setters concepts that were applied here, we will see later. The relevant thing here is known about the definition of class and object.

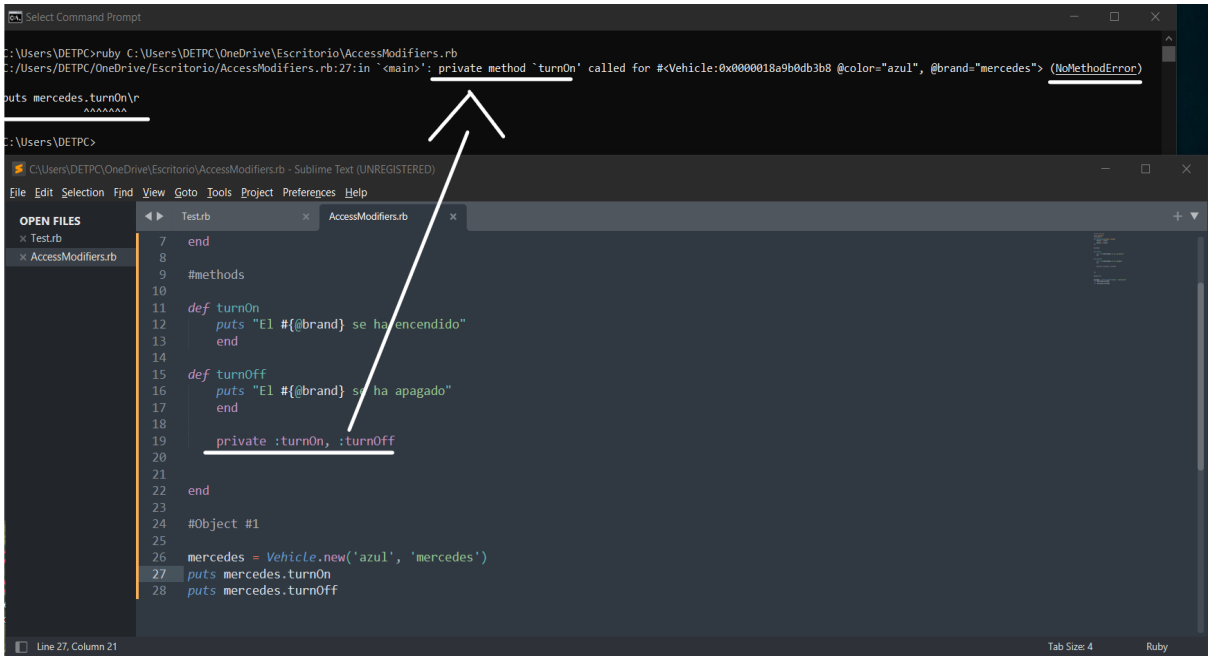
## 5.Access Modifiers

In Ruby, we have the following access modifiers to be used on methods.

1. **Public:** In this all members are available to everyone to modify.
2. **Private:** In this the members can only be accessed by functions inside the class.
3. **Protected:** The members in protected class can only be accessed by functions inside subclass. We will see the subclass concept in Inheritance.

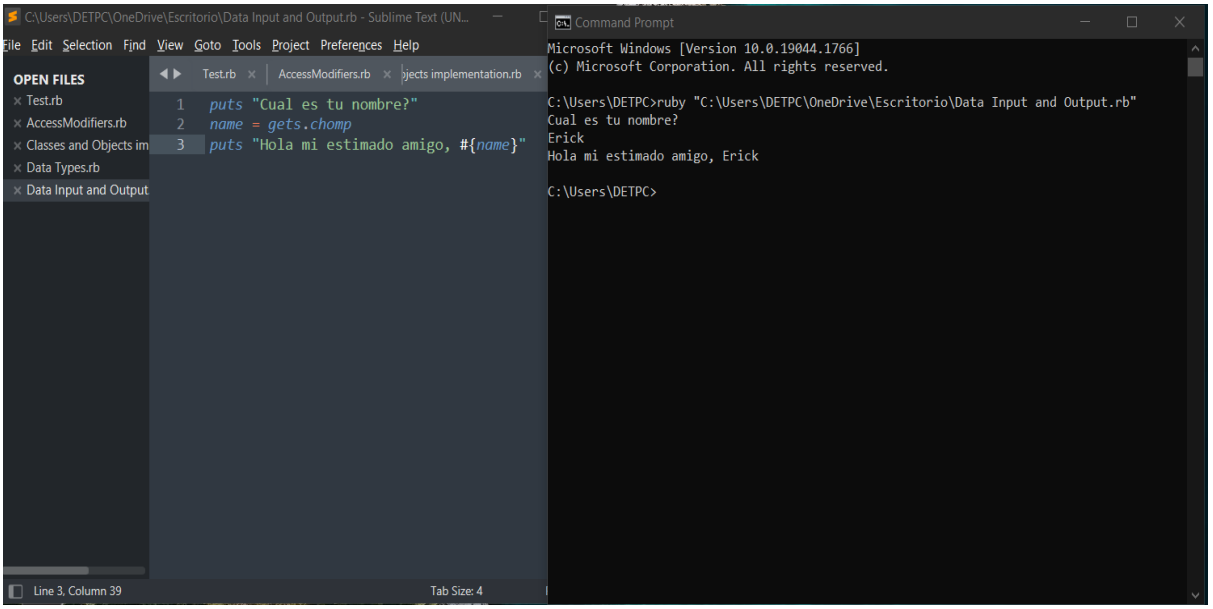
```
C:\Users\DETPC>ruby C:\Users\DETPC\OneDrive\Escritorio\AccessModifiers.rb
El mercedes se ha encendido

El mercedes se ha apagado

C:\Users\DETPC>
```

```ruby
#Class Vehicle
class Vehicle
#attributes
def initialize(color, brand)
    @color = color
    @brand = brand
end

#methods

def turnOn
    puts "El #{@brand} se ha encendido"
    end

def turnOff
    puts "El #{@brand} se ha apagado"
    end

    public :turnOn, :turnOff


end

#Object #1

mercedes = Vehicle.new('azul', 'mercedes')
puts mercedes.turnOn
puts mercedes.turnOff
```




## 6. Data Input and Output

An example program in Ruby that asks for your name and then says it:



## 7. Exceptions:

Exception handling in any programming language is very important, because if an error occurs in the code, an exception must be handled to prevent the program from working correctly.

To handle exceptions and avoid closing our program we use the **Begin** and **Rescue** keywords.

```
1  class Persona
2      def initialize()
3      end
4
5      def simulate_error()
6
7          begin
8
9              puts "Empezar el metodo"
10             raise "ERROR SIMULADO"
11
12         rescue
13             puts "Manejando el error"
14
15         end
16     end
17 end
18
19 p = Persona.new
20 p.simulate_error
21
```

```
C:\Windows\System32\cmd.exe

C:\Users\Andres\Desktop\RubyProjects\POO>EXCEPCIONES.rb
Empezar el metodo
Manejando el error

C:\Users\Andres\Desktop\RubyProjects\POO>
```

To better understand the example, line #10 simulates an error that would terminate the execution of the program with the keyword **raise**, an exception which causes a *RuntimeError*.

**Rescue** contains the action to take in case an exception occurs.

## 8. Data persistence

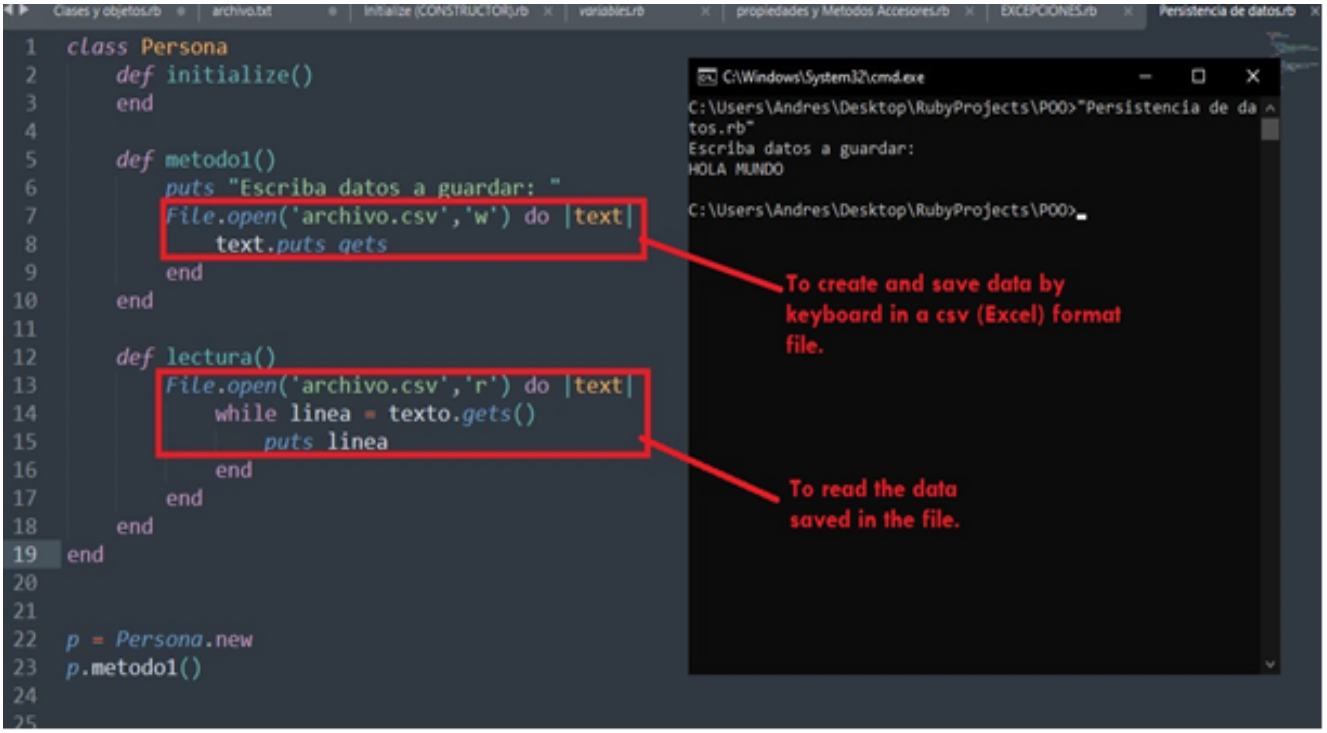
Persistence is the ability to save information from a program so that it can be used again at a later time. It is what users know as Save the file and then Open the file.

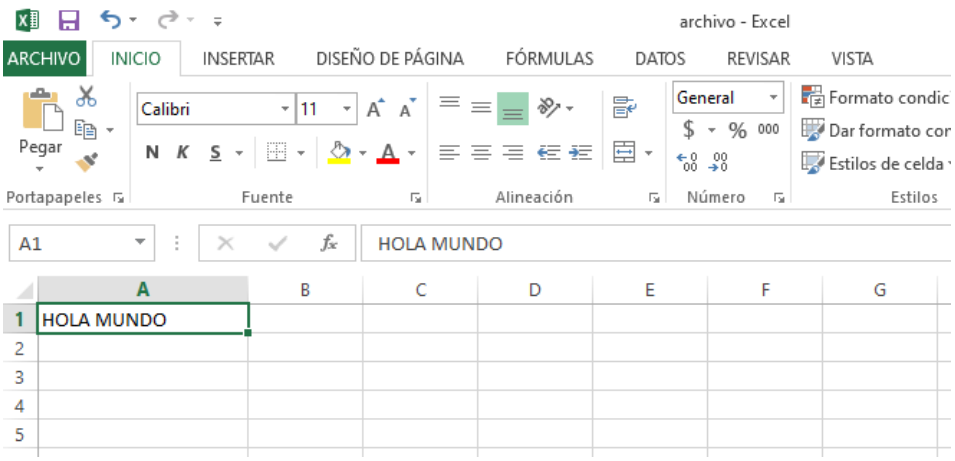
In order to create files in Ruby, we use the reserved word **File** and its **open** method with two parameters, one in which the file to be created is named along with the file type (.txt, .csv, etc) and in the second parameter a letter that represents what action can be performed with the file

w = write , r = read

```
File.open('archivo.csv','w') do |text|
```

text = variable which helps us to save the data we want to enter in a line.
do = to give the order to do the defined action.




## 9. Encapsulation

Object-oriented programming has certain characteristics, such as encapsulation. Here are some characteristics of encapsulation:

- Expose the external behavior of the class using methods.
- Protect internal data.
- Scopes in Ruby: *public, protected, private.*
- Accessories.

**Types of Accessories**:

attr_reader
attr_writer
attr_accessor

```ruby
class Ventilador
  attr_accessor :marca
  def initialize(params)
    @marca = params[:marca]
  end

  def presentacion
    "La marca del Ventilador es #{@marca}"
  end
end

obj = Ventilador.new(marca: 'XYZ')
puts obj.marca
obj.marca = 'ABC'
puts obj.marca
```

```
C:\Windows\System32\cmd.exe

C:\Users\Andres\Desktop\encap>attr_accessor.rb
XYZ
ABC

C:\Users\Andres\Desktop\encap>
```

```ruby
class Ventilador
  attr_reader :marca, :velocidad
  def initialize(params)
    @marca = params[:marca]
    @velocidad = 0
  end
end

obj = Ventilador.new(marca: 'XYZ')
puts obj.marca
puts obj.velocidad
```

```
C:\Windows\System32\cmd.exe

C:\Users\Andres\Desktop\encap>attr_reader.rb
XYZ
0

C:\Users\Andres\Desktop\encap>S
```

Access modifiers are the access that we can give to different methods, these can be:

- **Public:** Default
- **Private:** They will only work in that class, they cannot be called from the object nor can they inherit a child class.
- **Protected:** They cannot be called once the object is created, but they will be able to inherit a child class.

By default, the attributes of an object in Ruby are private, that is, they can only be accessed by methods of the same class. To give them visibility from other parts of the application, *attr_accessor (read and write)*, *attr_reader (read)*, and *attr_writer (write)* are used.

**Private:**

```ruby
class Food
  def public_method

  end
  private

  def bacon
  end

  def orange
  end

  def coconut
  end

end

```

**Protected:**

```ruby
class Food
  def initialize(name)
    @name = name
  end
  def ==(other)
    name == other.name
  end
  protected
  attr_reader :name
end
food = Food.new("chocolate")
puts food == food
```

## 10. Constructor

It has the characteristic that it is a method that is executed when we create an object of a class and is usually used to initialize values that the object will have.

In Ruby it is called as **initialize**.

```ruby
class Video

    attr_accessor :minutes, :tittle

    def initialize(tittle)
        self.tittle = tittle
        puts "I am the constructor"
    end

    def play
    end

    def pause
    end

    def stop
    end

end


video_POO = Video.new("Class and Object")
```

```
C:\Windows\System32\cmd.exe

C:\Users\Andres\Desktop\RubyProjects\POO>"Initialize (CONSTRUCTOR).rb"
I am the constructor

C:\Users\Andres\Desktop\RubyProjects\POO>
```

## 11. Getter, setter methods:

They are important methods in OOP since they allow us to manipulate the attributes or characteristics that our classes present.

A "setter" method is used to "load a value" (assign a value to a variable).

A "getter" method is used to "return the value" (only return the attribute information to the requester).

```ruby
class Celular

    def initialize(marca,color)
        @marca = marca
        @color = color
    end

    def llamar()
        puts @marca + "-" + @color + " esta llamando ..."
    end

    def marca
        return @marca
    end

    def marca=(marca)
        @marca = marca
    end
end
```

get

set

```
celular.rb   main.rb
1  require_relative "celular.rb"
2
3  c1 = Celular.new("Realme", "Negro")
4  puts c1.marca()
5  puts c1.marca = "Xiaomi"
6  #puts c1.color()
7  #puts c1.color = "Blanco
8  gets()
```

```
C:\Windows\System32\cmd.exe - main.rb
Microsoft Windows [Versión 10.0.19044.1288]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Andres\Desktop\RubyProjects\POO>main.rb
Realme
Xiaomi
```

## 12. Arrays and collections:

In Ruby there is the Array data structure that allows you to store a collection of data and then access any of its components by means of a subscript. To define an array, the nomenclature of square brackets [ ] is used, it can also contain different data types such as string, floating and boolean data types.

```
celular.rb   EXCEPCIONES.rb   main.rb   ARRAY.rb
1  class MyClass
2
3      def initialize()
4      end
5
6      def evaluar()
7          x = [1,2,3,"Andres",true]
8          puts x
9      end
10 end
11 clase = MyClass.new()
12 clase.evaluar()
13 gets()
```

```
C:\Windows\System32\cmd.exe - ARRAY.rb
C:\Users\Andres\Desktop\RubyProjects\POO>ARRAY.rb
1
2
3
Andres
true
```

Array

### Polymorphism

In Ruby we can achieve polymorphism using method overriding. In polymorphism, Ruby invokes the actual method during running of the program. One way of achieving polymorphism is with inheritance.

### Data Abstraction in Modules

modules are defined as a set of methods, classes, and constants together. For example, consider the sqrt() method present in the math module. Whenever we need to calculate the square root of a non-negative number, we simply call the sqrt() method present in the math module and send the number as a parameter without understanding the actual algorithm that actually calculates the square root of the numbers.

## Data abstraction in classes

We can use classes to perform data abstraction in ruby. The class allows us to group information and methods using access specifiers (private, protected, public). The Class will determine what information must be visible and what must not.

### Data abstraction using access control:

There are three levels of access control in Ruby (private, protected, public).

```
# Ruby program to demonstrate Data Abstraction

class Geeks

    # defining publicMethod

    public

    def publicMethod
        puts "In Public!"
        # calling privateMethod inside publicMethod
        privateMethod
    end

    # defining privateMethod

    private

    def privateMethod
        puts "In Private!"
    end
end


# creating an object of class Geeks
obj = Geeks.new

# calling the public method of class Geeks
obj.publicMethod
```

### Ruby interface with raise

This way would be using a module with empty methods whose body is to raise an exception when sent to objects.

```ruby
module Animal
  def eat
    raise 'Not implemented'
  end

  def travel
    raise 'Not implemented'
  end
end
```

```ruby
class Cat
  include Animal

  def eat
    puts 'The cat is eating'
  end
end
```

**Ruby interface with tests**

An interface can be imitated using unit tests. The way is that the tests would have to pass so that the code has an adequate level of validity, that is, the contract between the class and the "interface" would be fulfilled.

```ruby
describe Cat do

  before { @cat = described_class.new }

  describe '#meow' do
    it 'meows' do
      expect(@cat.meow).to be_a(String)
    end
  end
end
```