

# Introducción a la P00 \_



# Introducción a objetos en ruby

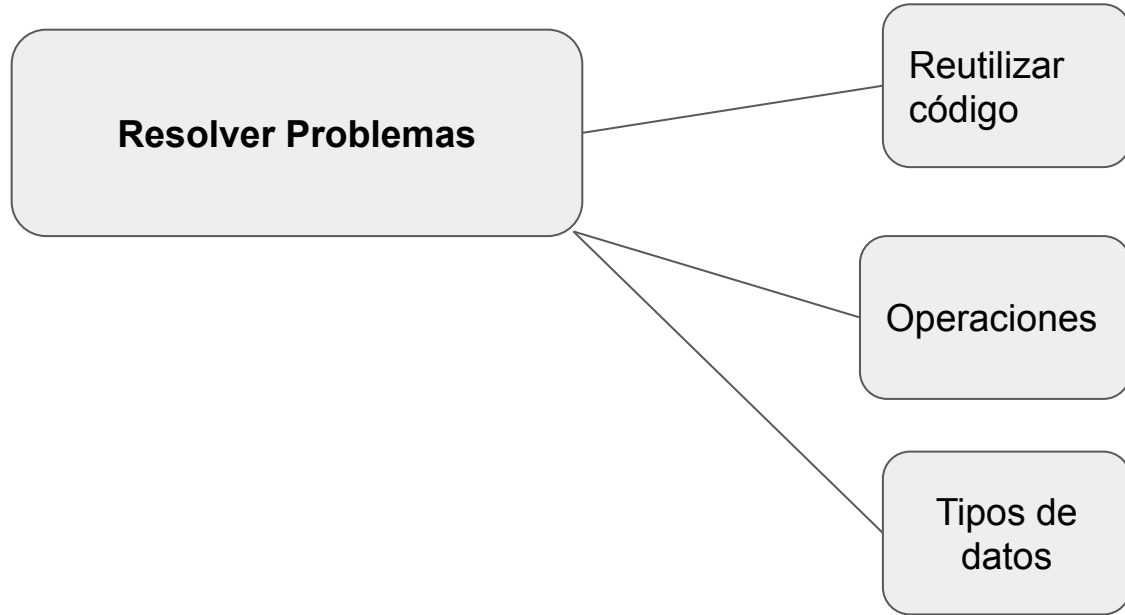
# Objetivos

- Conocer los conceptos de clase y objeto
- Crear clases
- Crear instancias de una clase
- Conocer las convenciones para nombrar las clases

## Breve historia de los objetos



# ¿Para qué sirve la programación orientada a objetos?



# Los tres conceptos más recurrentes

```
class Persona  
end
```

Clase

```
persona = Persona.new
```

Instanciar

Instancia



# Creando nuestra primera clase

```
class MiPrimeraClase  
end
```

# Instanciando nuestra primera clase

```
a = MiPrimeraClase.new
```

```
a.class # MiPrimeraClase
```



**UML**

# Objetivos

- Introducir los diagramas UML

# Diagramas de Clase

UML especifica muchos tipos de diagrama, nosotros trabajaremos particularmente con uno llamado diagrama de clases. Este diagrama se utiliza para especificar todas las clases que tenemos que construir.



## Y a código

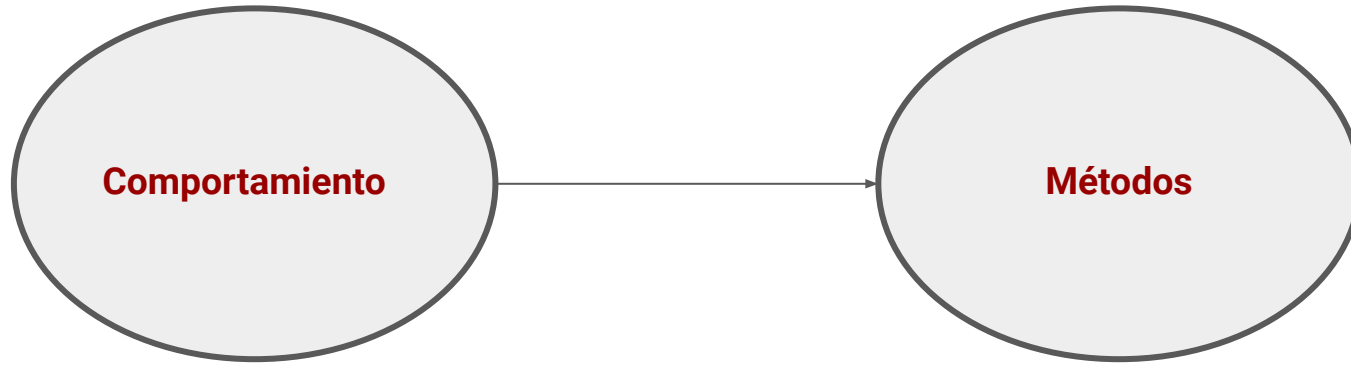
```
class MiPrimeraClase  
end
```

**Agregando comportamiento a los  
objetos**

# Objetivos

- Entender que son los métodos de instancia
- Agregar métodos de instancia a un objeto
- Utilizar métodos de instancia de un objeto
- Conocer los errores al llamar a un método de instancia desde main o directamente sobre la clase.
- Ver en un diagrama UML el comportamiento de un objeto Conocer el principio de abstracción

# Objetivos



- **Función: Permiten realizar acciones**

# Agregando un método de instancia

```
class Persona
  def saludar
    puts "hola!!"
  end
end
```



# Utilizando un método definido dentro de una clase

```
ignacio = Persona.new  
ignacio.saludar => "hola!!"  
ignacio.saludar => "hola!!"
```

# Intentando llamar al método de instancia desde otros contextos

- Intentando desde main

```
saludar() # undefined method `saludar' for main:Object
```

- Intentando directamente sobre una clase

```
Persona.saludar # undefined method `saludar' for Persona:Class
```

**Se llaman métodos de instancia porque deben ser ocupados sobre instancias.**

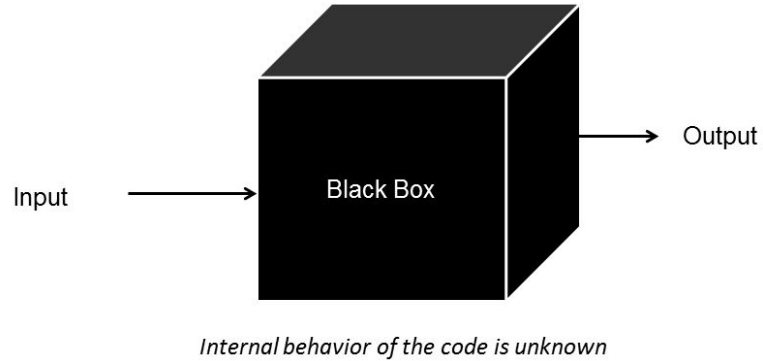
# Los métodos en UML



## Ejercicio

- Construir la clase Perro con el método ladrar.
- Al llamar al método se debe mostrar en pantalla bark.
- Instanciar 2 perros y hacerlos ladrar.

# El principio de abstracción



# Resumen

```
class Persona
  def saludar # Método de instancia
    puts "hola!!"
  end
end

p1 = Persona.new
p1.saludar
```

# Los estados de un objeto



# Objetivos

- Agregar estados a un objeto.
- Utilizar variables de instancia para agregar estados a un objeto.
- Conocer el principio de encapsulación.
- Representar los estados en un diagrama UML.

# Guardando un estado

```
class Vehiculo
  def encender()
    @encendido = 'on'
  end
  def apagar()
    @encendido = 'off'
  end
  def estado()
    @encendido
  end
end
```

## Guardando un estado(Ejemplo)

```
a1 = Vehiculo.new
```

```
a2 = Vehiculo.new
```

```
a1.apagar
```

```
a1.estado # => "off"
```

```
a2.encender
```

```
a2.estado # => "on"
```

# Alcance de las variables

Alcance de variables locales

```
def foo
  a=5
end
def bar
  puts a # undefined local variable or method `a' for main:Object
end
```

# Alcance de las variables

Alcance de variables de instancia

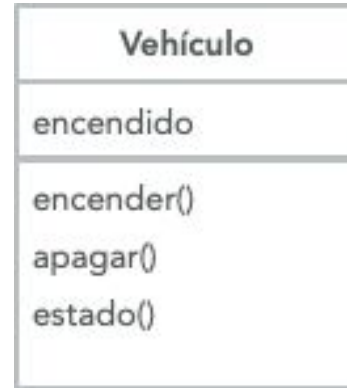
```
class Test
  def foo
    @a = 5
  end

  def bar
    puts @a
  end
end

test = Test.new
test.foo
test.bar # 5
```

# Representando los estados en UML

```
class Vehiculo
  def encender()
    @encendido = :on
  end
  def apagar()
    @encendido = :off
  end
  def estado()
    @encendido
  end
end
```



# De UML a código

```
class Persona
  def saludar()
    puts "hola"
  end
end
```



# Las variables de instancia se definen y ocupan dentro de los métodos de instancia

```
class Vehiculo
  def encender()
    @encendido = on
  end
  def apagar()
    @encendido = off
  end
  def estado()
    @encendido
  end
end

a1 = Vehiculo.new
a1.encendido # => undefined method `encendido' for #
<Vehiculo:0x00007fa676129728>
# Did you mean?  encender
```



# De UML a código

Persona
nombre
edad
saludar()

```
class Persona
  def saludar()
    puts "hola"
  end
end
```

# Principio de encapsulación

# Objetivos

- Conocer el principio de encapsulación
- Crear métodos getters y setters manualmente
- Crear métodos getters y setters utilizando los métodos `attr_reader`, `attr_writer` y `attr_accessor`

# Intentando violar el principio de encapsulación

```
class Mascota
  def initialize(nuevo_nombre)
    @nombre = nuevo_nombre
  end
end

m1 = Mascota.new("Shadow")
m1.nombre # => NoMethodError: undefined method `nombre' for
          #<Mascota:0x007ff7e38b84f8 @nombre="Shadow">
```

# Creando getters y setters manualmente

```
class Mascota
  def get_nombre #Método getter
    @nombre
  end
  def set_nombre(nombre) #Método setter
    @nombre = nombre
  end
end

m1 = Mascota.new
m1.set_nombre "Wishbone"
m1.get_nombre # => Wishbone
```

# Creando getters y setters con los nombres de la variable

```
class Mascota
  def nombre
    @nombre
  end
  def nombre=(nombre)
    @nombre = nombre
  end
end
```

```
m1 = Mascota.new
m1.nombre="Spike"
# O más fácil de leer
m1.nombre = "Spike"
```

# Creando getters y setters con attributes accessors

```
class Caja
  attr_accessor :ancho
end

c = Caja.new

c.ancho = 2 # Utilizamos el setter creado automáticamente
c.ancho # => 2 # Utilizamos el getter creado automáticamente.

Caja.instance_methods
# => [:ancho, :ancho=, :instance_of?, :public_send,
      :instance_variable_get...]
```

# Definiendo múltiples getters y setters

```
class Caja
  attr_accessor :ancho, :alto
end
```

```
class Caja
  attr_accessor :ancho
  attr_accessor :alto
end
```



# Creando getters y setters manualmente

```
class Experimento
  attr_reader :a # Define solo un método getter
  attr_writer :b # Define solo un método setter
end

exp = Experimento.new
exp.a # funciona
exp.a = 5 # Error
exp.b # Error
exp.b = 5 # funciona
```

## Ejercicio

- Simplificar el código ocupando attr\_reader , attr\_writer y attr\_accessor según corresponda.

```
class Empresa
  def nombre
    @nombre
  end

  def direccion
    @direccion
  end

  def direccion=(direccion)
    @direccion = direccion
  end
end
```

# Resumen

- Abstracción
- Encapsulación

# El método constructor

# Objetivos

- Crear objetos con valores iniciales
- Asignar valores iniciales al momento de crear un objeto

# Introducción

```
Array.new() # => []
```

```
Array.new(3, 3) # => [3, 3, 3]
```

# El método initialize

```
class Ejemplo
  def initialize(a)
    puts a
  end
end

Ejemplo.new(5) # => 5
Ejemplo.new('hola') # => hola
```

# Utilizando initialize para dar valores iniciales

```
class Semaforo
  def initialize(estado)
    @estado = estado
  end
end

s1 = Semaforo.new(:rojo) # => <Semaforo:0x00007f9eda8c6560 @estado=:rojo> s2
= Semaforo.new(:verde) # => <Semaforo:0x00007f9eda9dbc20 @estado=:verde>
```



# Utilizando initialize para dar valores iniciales

```
class Semaforo
  def initialize(estado)
    @estado = estado
  end
end

s1 = Semaforo.new(:rojo)
```

estado = :rojo



# De código a UML

```
class Persona
  def initialize()
    @nombre = ""
    @edad = 0
  end
  def saludar()
    puts "hola"
  end
end
```



# Ejercicio

Se pide crear la clase mascota, esta recibirá un nombre al momento de la construcción, se debe agregar setters y getters para poder utilizar y modificar ese nombre posteriormente.

# Constructores con argumentos con valores por defecto

```
class Casa
  def initialize(pisos = 1)
    @pisos = pisos
  end
end

casa1 = Casa.new #El valor de @pisos es 1
casa2_pisos = Casa.new(2) #El valor de @pisos es 2
```

# Resumen

```
class Persona
  attr_accessor :nombre
  def initialize(nombre)
    @nombre = nombre
  end
end
```

getter y setter

constructor

variable local

instancia

variable de instancia

# Ejercicios resueltos

# Ejercicio

Se pide crear la clase producto, un producto tiene nombre y stock, por defecto si el stock no se especifica será cero, el nombre se define al momento de crearlo.

# Ejercicio

- Se pide crear una clase punto para representar puntos dentro de un mapa, los puntos tendrán coordenada x y una coordenada y
- Se pide que la clase punto debe poder ser inicializada en cualquier posición. La posición de un punto puede ser vista fuera de la clase pero no debe poder ser cambiada desde fuera.
- Se pide que la clase punto tenga un método avanzar que permita incrementar la coordenada x en una unidad.



# Reutilizando código

# Objetivos

- Cargar objetos desde otros archivos
- Diferenciar el método `require` de `require_relative`

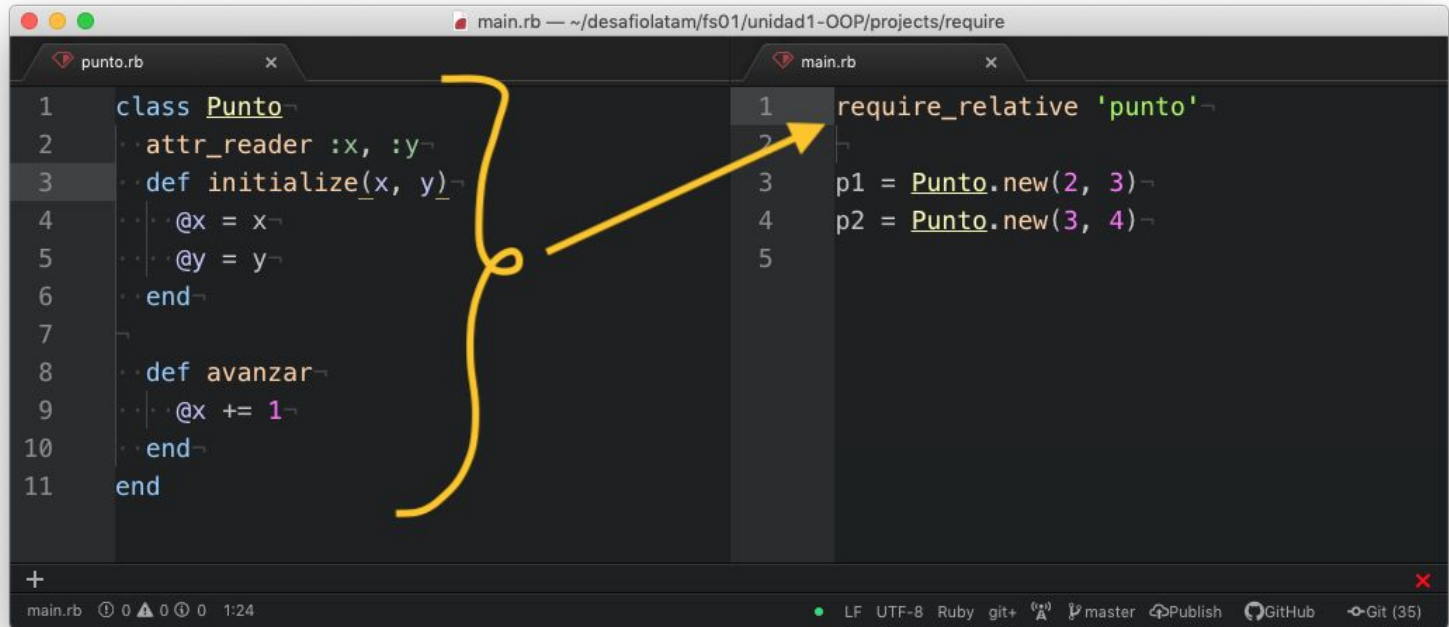
# La instrucción require\_relative

```
class Punto
  attr_reader :x, :y
  def initialize(x, y)
    @x = x
    @y = y
  end
  def avanzar()
    @x += 1
  end
end
```

```
require_relative punto

p1 = Punto.new(2,3)
p1.avanzar
```

# La instrucción require\_relative



```
main.rb — ~/desafiolatam/fs01/unidad1-OOP/projects/require
```

```
punto.rb
1 class Punto
2   attr_reader :x, :y
3   def initialize(x, y)
4     @x = x
5     @y = y
6   end
7
8   def avanzar
9     @x += 1
10  end
11 end
```

```
main.rb
1 require_relative 'punto'
2
3 p1 = Punto.new(2, 3)
4 p2 = Punto.new(3, 4)
5
```

The screenshot shows a code editor with two tabs: `punto.rb` and `main.rb`. The `punto.rb` tab contains a Ruby class definition for `Punto` with attributes `@x` and `@y`, and methods `initialize` and `avanzar`. The `main.rb` tab contains a `require_relative 'punto'` statement followed by two instances of `Punto` being created. A yellow bracket on the left side of the `punto.rb` tab groups the class definition, and a yellow arrow points from this bracket to the `require_relative 'punto'` line in the `main.rb` tab, illustrating the relationship between the two files.

# Ejercicio

El objetivo es separar la clase Punto del uso de la clase. Para ello se utilizará el ejercicio de los puntos

.

# Separando los archivos

El objetivo es separar la clase Punto del uso de la clase. Para ello se utilizará el ejercicio de los puntos .

```
class Punto
  attr_reader :x, :y
  def initialize(x, y)
    @x = x
    @y = y
  end
  def avanzar()
    @x += 1
  end
end
```

- Luego, para cargar el proyecto: Require  
Require\_relative

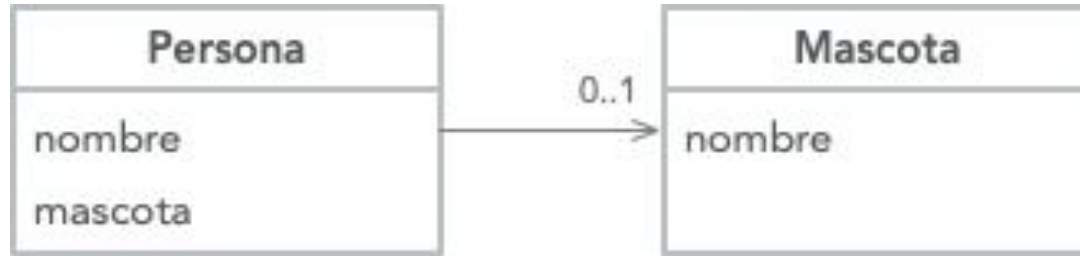
# Asociaciones

# Objetivos

- Trabajar con objetos que se asocian
- Leer diagramas UML con asociaciones
- Crear multiples clases y asociarlas
- Crear asociaciones con cardinalidad 0..1
- Crear asociaciones con cardinalidad 0..n



# Asociaciones



- Cardinalidad y dirección

# Traspassando el diagrama a código

```
class Persona
  attr_accessor :nombre, :mascota
end

class Mascota
  attr_accessor :nombre
end
```

# Traspasando el diagrama a código

```
class Persona
  attr_accessor :nombre, :mascota
  def initialize(nombre, mascota)
    @nombre = nombre
    @mascota = mascota
  end
end
```

```
class Mascota
  attr_accessor :nombre
  def initialize(nombre)
    @nombre = nombre
  end
end

mascota1 = Mascota.new("Fido")
persona1 = Persona.new("Fernanda",
                        mascota1)
```

# Revisando cardinalidad nula

```
persona1 = Persona.new("Fernanda")  
# wrong number of arguments (given 1, expected 2)
```

# Habilitando cardinalidad 0

```
class Persona
  attr_accessor :nombre, :mascota
  def initialize(nombre, mascota = nil)
    @nombre = nombre
    @mascota = mascota
  end
end

persona1 = Persona.new("Flavio")
persona1.mascota.nil? # true
```

# Asociaciones con cardinalidad n

```
class Persona
  attr_accessor :nombre, :mascotas
  def initialize(nombre, mascota = nil)
    @nombre = nombre
    @mascotas = []
    @mascotas.push mascota
  end
end
```

```
m1 = Mascota.new('Seymour')
p1 = Persona.new('Fry', m1)
m2 = Mascota.new('Nibbler')
p1.mascotas.push m2

#=> [#<Mascota:0x00007f87e88c8378
@nombre="Seymour">, #<Mascota:0x00007f
87e88c8210 @nombre="Nibbler">]
```

# Asociaciones con cardinalidad n

```
class Persona
  attr_accessor :nombre, :mascotas
  def initialize(nombre, mascotas)
    @nombre = nombre
    @mascotas = mascotas
  end
end

class Mascota
  attr_accessor :nombre
  def initialize(nombre)
    @nombre = nombre
  end
end
```

```
m1 = Mascota.new('Seymour')
m2 = Mascota.new('Nibbler')
p1 = Persona.new('Fry', [m1, m2])

# => <Persona:0x00007f87ea0d8300
# @nombre="Fry",# @mascotas=[
# <Mascota:0x00007f87ea0d8558
# @nombre="Seymour">,#<Mascota:0x00007f87ea
# 0d83c8 @nombre="Nibbler">
# ]>
```

# Ejercicio

- En el ejercicio anterior hicimos una suposición, que el constructor inicial solo recibía una mascota, pero esto no necesariamente debe ser así, dependerá de requerimientos específicos del sistema que siempre debemos consultar.
- Para contrastar repetiremos el ejercicio asumiendo lo contrario. el constructor de Persona debe recibir un arreglo de mascotas



# Ejercicio

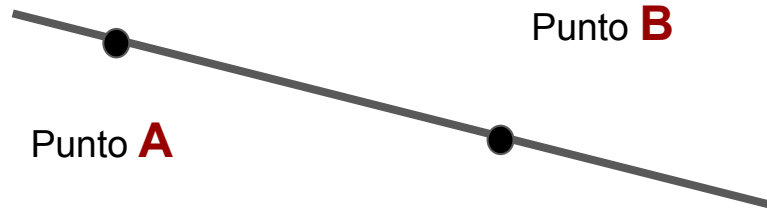
- Una persona puede tener múltiples redes sociales
- Las personas tienen un nombre y edad,
- Tipo de red social, pueden ser:

Facebook  
Instagram  
Pinterest  
Twitter  
Linkedin



# Ejercicio

- Se pide crear la clase recta,
- Considerando que una recta está construida a partir de dos puntos



# Introducción a la identidad

# Objetivos

- Introducir el concepto de identidad
- Diferenciar objetos a partir de su identificador

# ¿Qué es identidad?



```
class Persona
  attr_accessor :nombre
  def initialize(nombre)
    @nombre = nombre
  end
end

p1 = Persona.new("Trinidad")
p2 = Persona.new("Trinidad")
```

## ¿Cómo podemos probar que son distintos?

```
persona1 = Persona.new("Trinidad")  
persona2 = Persona.new("Trinidad")  
persona2.nombre = "Javiera"  
puts persona1.nombre # => Trinidad
```

# Ejemplo

- Construir una clase llamada persona
- Debe tener un nombre
- Considerar cantidad de km caminados.
- Toda persona parte con 0 km caminados.
- Se debe instanciar dos personas a partir de esta clase.



# Solución Ejemplo

- Se instanciaron dos personas,
- Ambas se llaman Javiera. Sin embargo no son la misma persona,
- Los kilómetros caminados por cada una de estas personas son distintos,
- Los kilómetros recorridos no se suman entre ellas,
- Formalmente se debe decir: son dos instancias distintas.





# ¿Cómo obtener el identificador de un objeto?

```
class Persona
end

p1 = Persona.new
p2 = Persona.new

puts p1.object_id #70351254781520
puts p2.object_id #70351254781500
```

# Ejemplo con Arrays

```
puts [1,2,3,4].object_id # 70110849102780
```

```
puts [1,2,3,4].object_id # 70110858362060
```

# Ejemplo con Strings

```
puts 'hola'.object_id # 70166159749440  
puts 'hola'.object_id # 70166159747080
```

# Ejemplo con Enteros

```
2.object_id # 5
```

```
2.object_id # 5
```

# Ejemplo con símbolos

```
:hola.object_id #1289948
```

```
:hola.object_id #1289948
```

# Identidad y variables

# Objetivos

- Trabajar con objetos y variables
- Conocer las implicancias de trabajar con el mismo objeto en dos o más variables.

# ¿Por qué es importante saber si dos objetos son distintos?

```
a = [1, 2, 3, 4]
b = a
a == b # true
```

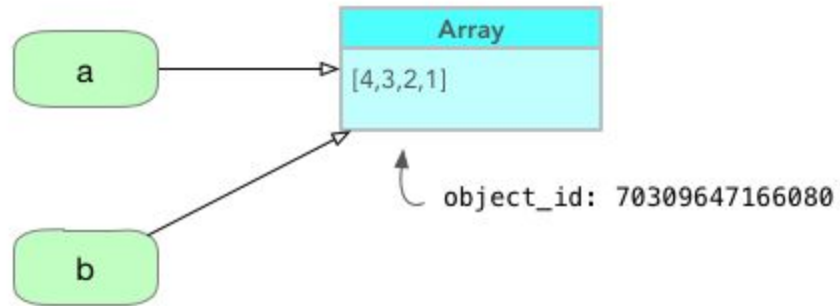
```
a = [1, 2, 3 , 4]
b = a

puts a.object_id == b.object_id # true

a[0] = 8
print b # [8, 2, 3, 4]
```

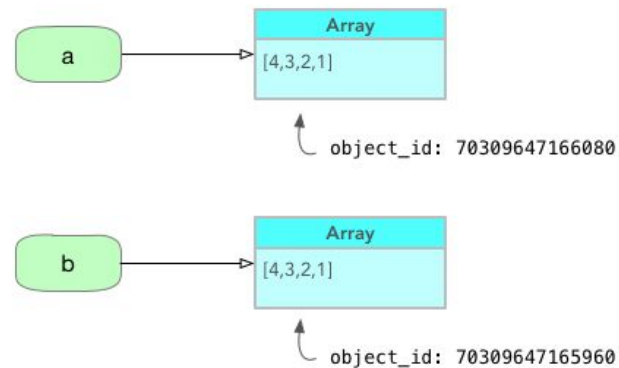


# Ilustrando lo sucedido

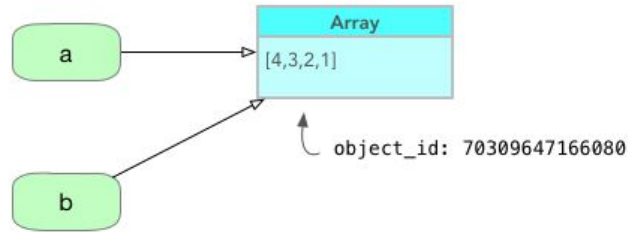


# Esto no sucede si trabajamos con objetos distintos

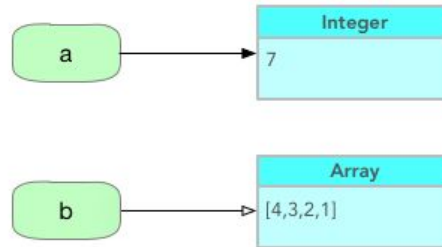
```
a = [1, 2, 3, 4]
b = [1, 2, 3, 4] # instanciamos un nuevo objeto
puts a.object_id == b.object_id # false
a[0] = 8
puts b # [1, 2, 3, 4]
```



# Asignar un nuevo valor a una variable no es lo mismo que modificar el objeto



```
a = 7
```



# Modificar V.S Asignar

```
arreglo[1] = 4  
persona.edad = 19
```

```
arreglo = [1,2,4]  
persona = Persona.new()
```

# Mutabilidad

# Objetivos

- Conocer los conceptos de mutabilidad e inmutabilidad
- Crear objetos cuyas operaciones devuelvan nuevos objetos.
- Crear métodos que modifiquen el estado de un objeto
- Crear métodos que no modifiquen el estado de un objeto.

# Introducción a mutabilidad

Un objeto es mutable si puede cambiar de estado.

En ruby la mayoría de los objetos son mutables, esto quiere decir que su estado (alguno de sus atributos) pueden cambiar.

```
class MoldeAuto
  def initialize()
    @color = "verde"
  end
end

m1 = MoldeAuto.new
#m1.@color sería un error
```

# Agregando un setter

```
class MoldeAuto
  attr_accessor :color #agregamos el setter y el getter simultáneamente
  def initialize()
    @color = "verde"
  end
end

m1 = MoldeAuto.new
m1.color = "rojo"
#m1.@color sigue siendo un error
```



# Agregando un setter

```
class MoldeAuto
  attr_accessor :color #agregamos el setter y el getter simultáneamente
  def initialize()
    @color = "verde"
  end
end

m1 = MoldeAuto.new
m2 = m1
m1.color = "rojo"
puts m2.color # => rojo
```

# No hablemos de objetos, hablemos de métodos

Methods
::[]
::new
::try_convert
#&
#*
#+
#-
#<<
#<=>
#==
#[]
#[]=
#abbrev
#assoc
#at
#bsearch
#clear
#collect
#collect!
#combination
#compact
#compact!
#compact

# Creando un método que modifique el estado

```
class Persona
  def initialize(nombre, caminado = 0)
    @nombre = nombre
    @caminado = caminado
  end
  def caminar(km = 1)
    # Aquí se modifica @caminado,
    #por lo que el método es mutable.
    @caminado += km
  end
  def caminado
    @caminado
  end
end
```

```
p1 = Persona.new("Javiera")
p2 = p1
p1.caminar(10)
puts p2.caminado
```

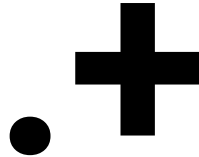
# Creando un método que modifique el estado

```
class Persona
  def initialize(nombre, caminado = 0)
    @nombre = nombre
    @caminado = caminado
  end
  def caminar(km = 1)
    Persona.new(@nombre, @caminado + km)
  end
  def caminado
    @caminado
  end
end
```

```
p1 = Persona.new("Daniel")
p2 = p1.caminar(10)

p2 =>#<Persona:0x00007fa1a89a7530
@nombre="Daniel", @caminado=10>
```

**Para recordar**



**Es un Método**

# Ejemplo

- Supongamos que tienen canastas
- Cada canasta se compone de cierta cantidad de frutas, velas aromáticas y/o tarjetas.
- Se pide crear la clase Canasta que reciba las cantidades de cada elemento y un método que suma la cantidad de elementos por separado y devuelva la cuenta total.
- Se pide además, agregar el método .+ para poder juntar una canasta con otra, este método debe devolver una canasta nueva con la suma de cada elemento por separado.



# Manejo de excepciones

# Objetivos

- Levantar excepciones
- Manejar excepciones levantadas



# Ejemplo de excepción

```
'string' + 2 # TypeError (no implicit conversion of Integer into String)
```

- Manejando excepciones

```
begin
  'no se puede sumar un string con un int' + 2
rescue
  puts 'pero estos errores pueden ser manejados'
end
puts 'y ahora el programa corre de forma normal'
```

# Levantando una excepción

```
2.5.3 :048 > raise
```

```
Traceback (most recent call last):
```

```
  2: from /Users/gonzalosanchez/.rvm/rubies/ruby-2.5.3/bin/irb:11:in `<main>'
```

```
  1: from (irb):48
```

# Rescatando una excepción

```
begin
  raise 'Soy un error'
rescue
  puts 'pero fui salvado'
end
puts 'y ahora el programa corre de forma normal'
```

# Mostrando la excepción en el rescate

```
begin
  raise 'Error tipo 409'
rescue StandardError => e
  puts 'Fui salvado'
  puts "aunque detecté el problema: #{e}"
end
puts 'y ahora el programa corre de forma normal'
```

# Introducción a tipos de Excepciones

```
begin
  raise 'Error'
rescue StandardError => e
  puts e.class # => RuntimeError
end
```

# ArgumentError

```
def method1(x, y)
  raise ArgumentError, 'x is not an Integer' if x.class != Integer
  raise ArgumentError, 'y is not an Integer' if y.class != Integer
end

method1(1, 'hola') # ArgumentError: y is not an Integer
```

# Mejorando un ejercicio

```
class Persona
  attr_accessor :nombre, :mascotas
  def initialize(nombre, mascota = nil)
    @nombre = nombre
    @mascota = nil
  end
end
```

```
class Mascota
  attr_accessor :nombre
  def initialize(nombre)
    @nombre = nombre
  end
end
```

## Mejorando un ejercicio(parte 2)

```
class Persona
  attr_accessor :nombre, :mascotas
  def initialize(nombre, mascota = nil)
    raise ArgumentError, "Argument mascota is of type #{mascota.class} but not Mascota" if
      mascota.class != nil || mascota.class != Mascota
    @nombre = nombre
    @mascota = nil
  end
end

p1 = Persona.new('Fry', 'nibler') #Argument mascota is of type String but not Mascota
```



**Cierre**

**{desafío}**  
**latam\_**

*Academia de  
talentos digitales*

[www.desafiolatam.com](http://www.desafiolatam.com)