



Introducción a Objetos

Requisitos

- Manejo y creación de Métodos
 - Métodos con parámetros
 - Retorno (y entender que es implícito)
- Entender el concepto de contexto Diferenciar variables locales / globales

Introducción a objetos en ruby

Objetivos:

- Conocer los conceptos de clase, instancia y objeto
- Crear clases
- Instanciar una clase
- Conocer las convenciones para nombrar de las clases

Introducción

La programación orientada a objetos es una forma distinta de pensar los problemas, que consiste en agrupar los elementos de un programa en torno a objetos. Estos objetos son entidades que se componen de atributos y comportamientos y gracias a esto podemos reutilizar fácilmente nuestro código y abstraernos de varios de los procesos.

Breve historia de los objetos

La programación orientada a objetos es bastante más antigua de lo que a veces pensamos. Nace conceptualmente a finales de los años 50, fue incorporado en el lenguaje de programación SIMULA a principio de los 60 y popularizado en los 70 con el lenguaje SmallTalk y en los 80 con C++

¿Para qué sirve la programación orientada a objetos?

La programación orientada a objetos permite crear tipos de datos propios así como también operaciones propias para resolver los problemas que se necesitan resolver. Además, permite reutilizar código propios o de un tercero con el objetivo de resolver problemas similares.

¿Por qué se dice que es difícil entender objetos?

Comúnmente escucharás decir a personas que estudiaron programación orientada a objetos en la universidad que es un tema difícil de comprender y puede ser que tengan un poco de razón.

La Programación orientada a objetos agrega varios conceptos nuevos a los que hemos aprendido hasta ahora, pero la principal dificultad a la hora de comprender la programación orientada a objetos es que no existe una fórmula universal para resolver problemas. La buena noticia, es que parte importante de resolver los problemas consiste en identificar identidades y agrupar los atributos en torno a estas identidades, proceso que es similar en la mayoría de los problemas y que se estudiarán en esta unidad.

Los tres conceptos más importantes

Los 3 conceptos más importantes de la programación orientada a objetos:

- Clase
- Instanciar
- Instancia (Objeto)

El proceso principal consiste en contruir o utilizar una clase existentes para crear varias instancias del mismo tipo. El proceso descrito como crear una de estas instancias se llama Instanciar.

La Clase

La clase es la base, se puede definir como un molde que permite construir varios objetos del mismo tipo. Por ejemplo la clase **Array** da la posibilidad de construir todos los arreglos que se requieran y todos ellos tendrán métodos como `.include?`, `.each` y `.map`.

A través de las clases se puede crear objetos nuevos.

Instancia u objeto

La instancia es el producto final o sea cuando escribimos `a = [2,3,4,5]` lo que queda guardado dentro de `a` es un objeto, en este caso esta instancia es del tipo array.

La instancia también recibe el nombre de objeto, por lo que podemos decir que una instancia es del tipo array o podemos decir que un objeto es del tipo array.

Instanciar

Instanciar es el acto de generar la instancia, cuando se hace `a = [2,3,4,5]` o `a = Array.new` lo que estamos haciendo es exactamente eso. Instanciar.

La forma más común de instanciar es utilizando el método `.new` pero algunos objetos como los arreglos y los hashes tienen una sintaxis especial.

Creando nuestra primera clase

Crearemos nuestra primera clase en IRB. Para crear una nueva clase utilizaremos la instrucción `class`.

```
class MiPrimeraClase  
end
```

Instanciando nuestra primera clase

Una vez creada la clase, se puede instanciar. Si recordemos la definición de clase como el molde, una vez que se define la clase se puede ocupar para crear todos los objetos que se necesiten.

```
a = MiPrimeraClase.new
```

Se puede observar que clase es un objeto utilizando el método `.class`

```
a.class # MiPrimeraClase
```

La clase que acabamos de crear no hace absolutamente nada. En un próximos capítulos le agregaremos atributos y comportamientos.

Convención Importante

Los nombres de las clases para distinguirlos de cualquier otro elemento del lenguaje deben empezar con una letra mayúscula (como si fueran una constante). Para distinguirlo de las constantes solo se ocupa la primera letra mayúscula mientras que las constantes suelen escribirse completamente en mayúsculas.

UML

Objetivos:

- Introducir los diagramas UML

Introducción a UML

- UML es un lenguaje para especificar, visualizar, construir y documentar artefactos de software. Artefactos como Objetos.
- UML es un lenguaje pictográfico o sea aprender UML significa aprender a leer y generar diagramas UML.

Diagramas de clases

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. Esta documentación es muy útil en los programas con varias clases, pues nos ayuda a ver de forma sencilla todas las componentes de un programa y cómo se relacionan.

Nuestra primera clase en UML

La clase más sencilla que podemos construir en UML es una clase que no tiene nada, como la que construimos en el capítulo anterior.



De UML a código

```
class MiPrimeraClase
end
```

Agregando comportamiento a los objetos

Objetivos:

- Entender los métodos de instancia como el comportamiento de un objeto
- 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

Introducción

Hasta el momento solo hemos creado clases vacías. En este capítulo aprenderemos a agregar comportamiento a los objetos.

¿Qué es un comportamiento?

El comportamiento de los objetos son los métodos. Estos son los que les permiten realizar acciones y por lo mismo se les dice comportamiento.

Ejemplo con arrays

Por ejemplo, los arrays pueden informar si tienen un elemento y cuántos elementos tienen, como en el siguiente ejemplo:

```
[1, 2, 3, 4].include? 2 # => true
[1, 2, 3, 4].size() # => 4
```

Estos tipos de métodos reciben el nombre de **métodos de instancia** y están definidos dentro de cada clase. Es por ello que en esta oportunidad se enseñará a crear métodos de instancias y a utilizar estos métodos.

Agregando un método de instancia

Para agregar un método de instancia lo que tenemos que hacer es definir el método dentro de la clase

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

```
:saludar
```

Utilizando un método definido dentro de una clase

Para utilizar el método de instancia tenemos que instanciar un objeto, una vez instanciado podemos ocupar sus métodos todas las veces que necesitemos.

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

Intentando llamar al método de instancia desde otros contextos

Cuando se define un método de instancia dentro de una clase, la instancia es el único lugar donde se puede ocupar.

Desde main

Si intentamos llamar al método de instancia que acabamos de definir desde main obtendremos un error.

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

Intentando directamente sobre una clase

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

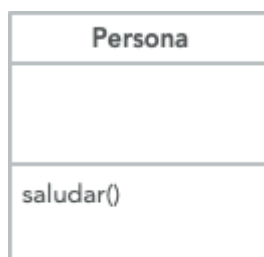
Este error hace mucho sentido porque se definió el método para la instancia y no para la clase.

Es tan fácil como recordar que se llaman métodos de instancia

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

Los métodos en UML

En un diagrama de clases de UML se especifican los nombres de los métodos y los parámetros que reciben. Por ejemplo si representamos en UML el objeto que acabamos de crear veremos:



En UML no se especifica el comportamiento, solo el nombre de este y se ocupa el tercer cuadrado para definirlos, en el segundo van los atributos estos los estudiaremos en el próximo capítulo.

Ejercicio resuelto:

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

Solución

```
class Perro
  def ladrar
    puts 'bark'
  end
end

sparky = Perro.new
fluffy = Perro.new
sparky.ladrar
fluffy.ladrar
```

```
bark
bark
```

El principio de abstracción

La programación orientada a objetos tiene diversos principios fundamentales, el primero es el principio de abstracción.

El proceso de abstracción consiste en contestar a la pregunta ¿Qué hace? En lugar del ¿Cómo lo hace?

Un ejemplo de esto que se vio anteriormente, es en el diagrama UML ya que se tiene el método saludar, esto solo significa que las personas saludan pero no se especifica el como saludan.

¿Para qué sirve el principio de abstracción?

La programación orientada a objetos tiene un fuerte foco en la reutilización. Cuando utilizamos un objeto lo utilizamos como una caja negra, y que no se conoce exactamente el proceso, pero sabe lo que hace y lo que se necesita.

Hay varios ejemplos como estos, cuando se ha hablado de arreglos, cuando se llama al método contar, ya que no se sabe exactamente cómo es el código de contar, pero si se sabe que devuelve la cuenta.

Cuando se deba programar objetos se tiene que pensar que serán utilizados como cajas negras, o sea, es importante que los nombres de los métodos sean descriptivos y que se mencione el propósito de estos. Lo último es obvio ya que es útil para que el programador los ocupe.

Resumen

Los objetos pueden tener comportamientos, esto se implementa a través de métodos que se definen dentro de la clase como en el siguiente código

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

Para utilizar un método de instancia se debe crear un objeto a partir de la clase.

```
p1 = Persona.new
p1.saludar
```

Es importante tener en consideración que cuando se tenga que trabajar con objetos, estos serán ocupados como una caja.

Los estados de un objeto

Objetivo:

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

Introducción

Los objetos pueden guardar contenido y este contenido puede ser modificado. Por ejemplo los arrays pueden guardar valores en su interior, un Hash puede guardar las claves y valores.

Para ser precisos, usaremos la palabra estado para referirnos al contenido y se utilizará variables de instancia para representar estos estados.

Guardando un estado

Para guardar estados de un objeto se ocupan las variables de instancia, estas se distinguen de otros tipos de variable porque empiezan con `@` y las variables de instancia deben ser definidas y utilizadas dentro de métodos de instancia.

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

Ahora con esta clase de vehículo podemos crear todos los que queramos y cada uno tendrá un estado encendido independiente de los otros.

Ejemplo:

```
a1 = Vehiculo.new
a2 = Vehiculo.new
a1.apagar
a1.estado
a2.encender
a2.estado
```

Podemos ver que mientras uno de los estados es encendido y el otro es apagado.

Alcance de las variables

Alcance de variables locales

Recordemos el concepto de alcance. En las variables locales cuando definimos una variable dentro de un método no podíamos acceder a ellas desde otro método.

```
def foo
  a = 5
end

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

foo()
bar()
```

Alcance de variables de instancia

El alcance de las variables de instancia es distinto, estas viven en conjunto con el objeto y son accesible a lo largo de todos sus métodos.

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

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

Entender el propósito es más importante que entender las reglas.

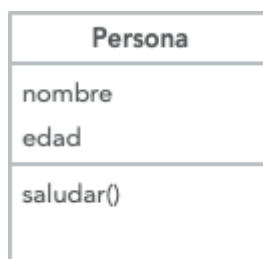
Más que aprender las reglas de memoria lo importante es entender el propósito, las variables de instancia sirven para reflejar estados de un objeto, por lo tanto es lógico que sus cambios perduren una vez terminado el método.

Representando los estados en UML

Los estados se agregan dentro del cuadrado que dejé en blanco en secciones anteriores. Si se toma como ejemplo el código del vehículo que se contruyó anteriormente:

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

Se obtiene el siguiente diagrama:



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

En ruby los atributos no se definen en un lugar específico, pueden estar definidos en cualquiera de los métodos, pero se debe tener en cuenta que hay que tener cuidado de no utilizarlos a través de un método, un atributo que todavía no ha sido definido, como por ejemplo:

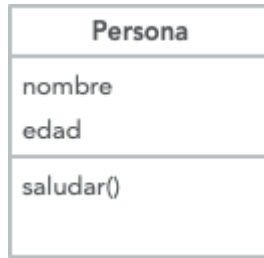
```
class Vehiculo
  def encender()
    @encendido = on
  end
  def apagar()
    @encendido = off
  end
  def estado()
    @encendido
  end
end

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

Más adelante hablaremos de un método llamado constructor que se carga al momento de instanciar los objetos y es el lugar por excelencia para darle valores iniciales a los atributos de un objeto.

De UML a código

¿Qué sucede si se intenta transformar directamente de UML a código?



Si se observa la clase definida, se tiene una idea básica de lo que se requiere hacer. Existe la clase llamada persona y un método saludar, por lógica se sabe que en alguna parte se necesitarán los atributos nombre y edad pero este ejemplo no dice nada al respecto. Por lo tanto, la mejor representación que se podría lograr por ahora es:

```
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`

Introducción

El principio de encapsulación es muy importante en la programación orientada a objetos. Esta dice que los estados internos deben estar protegidos del exterior.

En ruby esta implementación es muy directa. Las variables de instancia no pueden ser accedidas desde fuera del objeto.

Intentando violar el principio de encapsulación

En Ruby todas las variables de instancia son privadas. Esto quiere decir que no pueden ser accedidas desde fuera de la clase.

Para demostrarlo en este ejemplo se intenta acceder desde afuera:

```
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">
```

Analizando el error

```
# => NoMethodError: undefined method `nombre' for
#<Mascota:0x007ff7e38b84f8 @nombre="Shadow">
```

El error también da una pista de lo que está sucediendo, dice undefined method `nombre` o sea Ruby supuso que estábamos preguntando por un método en lugar de una variable y esto se debe a que no se pudo haber preguntado por algo mas, por ello hay que tener en cuenta que solo los métodos pueden ser llamados desde fuera del objeto.

Principio de encapsulación

En resumen: Los estados internos de un objeto deben estar protegidos y solo deben ser accedidos y modificados a través de métodos.

Introducción a métodos getters y métodos setters

Cuando se necesita acceder a un estado (variable de instancia) de un objeto, se tiene que crear métodos. Estos métodos son tan comunes y conocidos que tienen sus propios nombres, se llaman **getters y setters**.

Existen diversas formas de hacer getters y setters, principalmente son 3 y se pueden definir de la siguiente forma, la mala, la buena y la mejor.

Creando getters y setters manualmente

Esta vendría siendo la forma mala, pero es la mas intuitiva:

```
class Mascota
  def get_nombre #Método getter
    @nombre
  end

  def set_nombre(nombre) #Método setter
    @nombre = nombre
  end
end
```

Al agregar los métodos `get_nombre` y `set_nombre` ahora se puede modificar el estado nombre de cualquiera de las mascotas.

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

Existe una forma mejor de definir los getters y setters que ahorra un poco de código a la hora de utilizarlos.

Creando getters y setters con los nombres de la variable

Una mejor forma de crear getters y setters es utilizar el nombre de la variable como nombre del método:

```
class Mascota
  def nombre
    @nombre
  end

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

Aquí vemos algo curioso, el = es parte del nombre del método, entonces al hacer

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

De esta forma da la idea de que estamos modificando directamente una variable dentro del objeto, pero realmente estamos ocupando un método setter para cambiarlo.

Creando getters y setters con attributes accessors

En ruby es posible definir de forma simultánea la variable, los getters y setters a través del `attr_accessor`:

```
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.
```

`attr_accessor` define los getters y setters de la misma forma que se vió previamente. Esto se puederevisar con el método `.methods`

```
Caja.instance_methods
# => [:ancho, :ancho=, :instance_of?, :public_send, :instance_variable_get,
:instance_variable_set, :instance_variable_defined?,
:remove_instance_variable, :private_methods, :kind_of?, :instance_variables,
:tap, :is_a?, :extend, :define_singleton_method, :to_enum, :enum_for, :<=>,
:==, :!=, :!~, :eql?, :respond_to?, :freeze, :inspect, :display, :send,
:object_id, :to_s, :method, :public_method, :singleton_method, :nil?, :hash,
:class, :singleton_class, :clone, :dup, :itself, :taint, :tainted?, :untaint,
:untrust, :trust, :untrusted?, :methods, :protected_methods, :frozen?,
:public_methods, :singleton_methods, :!, :==, :!=, :__send__, :equal?,
:instance_eval, :instance_exec, :__id__]
```

Dentro de la lista se encuentran los métodos `ancho` y `ancho=` Estos se van agregando automáticamente gracias a `attr_accessor`.

Definiendo múltiples getters y setters

Muy frecuentemente se tendrá que definir más de un getter y setter. Aquí existen dos opciones que son correctas:

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

o

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

Definiendo getters y setters manualmente

Es posible que se necesite que un atributo tengan únicamente un getter o únicamente un setter, esto puede ser útil para indicarle a otros programadores que cierto atributo de nuestro objeto no debe ser modificado directamente o que existe una mejor forma de leerlo a través de otro método.

Si bien podemos agregarlos directamente como se explicó al principio del capítulo , cabe mencionar que existen métodos específicos para hacerlo de forma rápida

```
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:

Un programador poco experimentado con Ruby nos entregó el siguiente código

```
class Empresa
  def nombre
    @nombre
  end

  def direccion
    @direccion
  end

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

La idea de esto es simplificar el código ocupando `attr_reader`, `attr_writer` y `attr_accessor` según corresponda.

Solución

El primer paso es identificar cuántos getters y setters hay. En este caso vemos nombre tiene un getter y dirección tiene un getter y un setter. Por lo que nuestro código quedaría así:

```
class Empresa
  attr_reader :nombre
  attr_accessor :direccion
end
```

Resumen

La programación orientada a objetos busca la reutilización de estos, para lograr este propósito hemos estudiado dos principios fundamentales:

- Abstracción
- Encapsulación

El principio de abstracción especifica que los objetos son cajas negras, su método nos especifica qué sucede pero no el como sucede.

El principio de encapsulación dice que no se puede acceder directamente a los estados de estas cajas solo podemos ocupar sus métodos para modificar los estados y esto ayuda a proteger el funcionamiento y evitar un mal uso accidental de estos.

El método constructor

Objetivos:

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

Introducción

En muchas situaciones se necesitará que los objetos tengan un valor inicial, ya sea siempre el mismo o uno que se asigne al momento de la construcción. Un ejemplo de esto utilizando el método `.new` para crear arreglos es el siguiente:

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

se puede crear un arreglo con varios elementos con:

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

Si bien esta forma no es la recomendada para crear arreglos, entrega una idea de como se ocupan los constructores.

Nota: Cuando creamos arreglos de la forma tradicional ej: `[1, 2, 3]` también se llama al constructor aunque sea menos transparente para nosotros.

El método initialize

Las clases al momento de instanciarse llaman automáticamente a un método llamado `initialize`.

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

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

Utilizando initialize para dar valores iniciales

El método initialize es frecuentemente utilizado para asignar valores iniciales. Para probar esto, se debe construir la clase semáforo que al momento de crear reciba el color que se encuentra inicialmente:

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

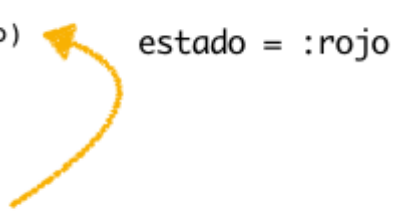
s1 = Semaforo.new(:rojo) # => <Semaforo:0x00007f9eda8c6560 @estado=:rojo>
s2 = Semaforo.new(:verde) # => <Semaforo:0x00007f9eda9dbc20 @estado=:verde>
```

Al igual que los métodos, una clase se puede leer desde el uso. La primera línea que debemos ver es

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

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

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



The diagram illustrates the execution flow. A yellow arrow points from the argument `:rojo` in the `new` method call to the `estado` parameter in the `initialize` method definition. Another yellow arrow points from the `estado` parameter to the `@estado = estado` assignment line, showing how the value is stored in the instance variable.

Al ejecutar `Semaforo.new` se está llamando automáticamente al método initialize de la clase Semaforo y le se le está pasando el argumento `:rojo`.

Dentro del método initialize se asigna el valor `:rojo` a la variable local `estado`. Pero hay que recordar que las variables locales mueren al terminar el método, si se quiere que persista hay que guardar esta variable dentro de una variable de instancia.

Es por esto es que se agregó la instrucción `@estado = estado`, para guardar el valor y poder seguir trabajando con el, una vez terminado el método constructor.

Ejercicio desarrollado:

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

Solución

Se pide crear la clase mascota, esta parte es exactamente igual que todas las otras.

```
class Mascota
end
```

Nos piden que se le puede asignar un valor inicial al momento de construirlo, aquí estamos hablando del constructor,

```
class Mascota
  def initialize(nombre)
    end
end
```

El nombre de la mascota tiene que persistir, por lo mismo tenemos que guardarlo en una variable de instancia.

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

Finalmente nos piden un método getter y setter, podemos ocupar attr_accessor para definirlo automáticamente.

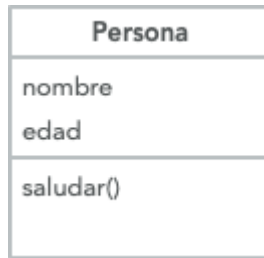
```
class Mascota
  attr_accessor :nombre
  def initialize(nombre)
    @nombre = nombre
  end
end
```

Ahora podemos crear directamente mascotas con nombre:

```
primera_mascota = Mascota.new("Spike")
primera_mascota.nombre # => Spike
```

De UML a código

Tenemos el siguiente ejercicio:



y como ya se conoce lo que son los constructores, se puede declarar valores iniciales al principio.

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

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

Constructores con argumentos con valores por defecto

En algunos casos requeriremos que el usuario ingrese un valor en el constructor pero si no lo ingresa es cuando asumiremos un valor. Por ejemplo vamos a crear la clase casa, que por defecto tendrá un piso pero al momento de instanciarse podría definirse otro valor.

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

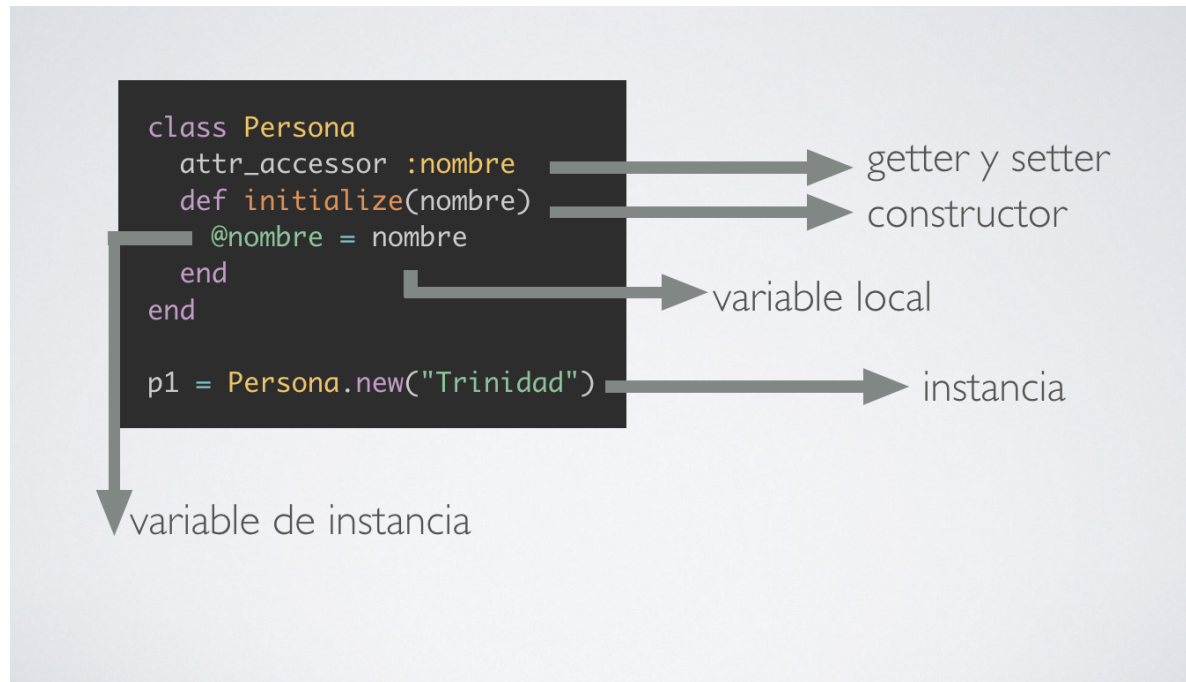
Al definirla de esta forma nosotros podemos crear casas de la siguiente forma:

```
casa1 = Casa.new
casa2_pisos = Casa.new(2)
```

El método initialize es un método como cualquier otro. La única diferencia es que se carga automáticamente cuando instanciamos un objeto.

Resumen

Antes de terminar el capítulo repasemos los conceptos importantes que hemos aprendido.



Ejercicios resueltos

Ejercicio Producto:

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

Solución

1) La clase producto:

```
class Producto
end
```

2) La clase producto tiene nombre y stock que se definen al momento de crearlo pero el stock por defecto es cero.

```
class Product
  def initialize(name, stock = 0)
    @name = name
    @stock = stock
  end
end
```

Ejercicio puntos

- 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 ser poder inicializada en cualquier posición
- La posición de un puntos 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.

Solución

1) Creando la clase punto

```
class Punto
end
```

2) Se pide que pueda ser inicializado en cualquier posición, obviamente esto hace referencia que al crear el objeto le tenemos que dar coordenadas , esto implica que se debe utilizar el método constructor.

```
class Punto
  def initialize(x, y)
    @x = x
    @y = y
  end
end
```

3) La posición de cualquiera de los puntos debe poder ser leída desde fuera de la clase pero no cambiada. Esto significa que debemos agregar un attr_reader pero **no** debemos agregar un attr_writer o attr_accessor

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

4) Creando el método avanzar que incremente en 1 la posición de x

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

Finalmente ya se puede probar la clase

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


Reutilizando código

Objetivos

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

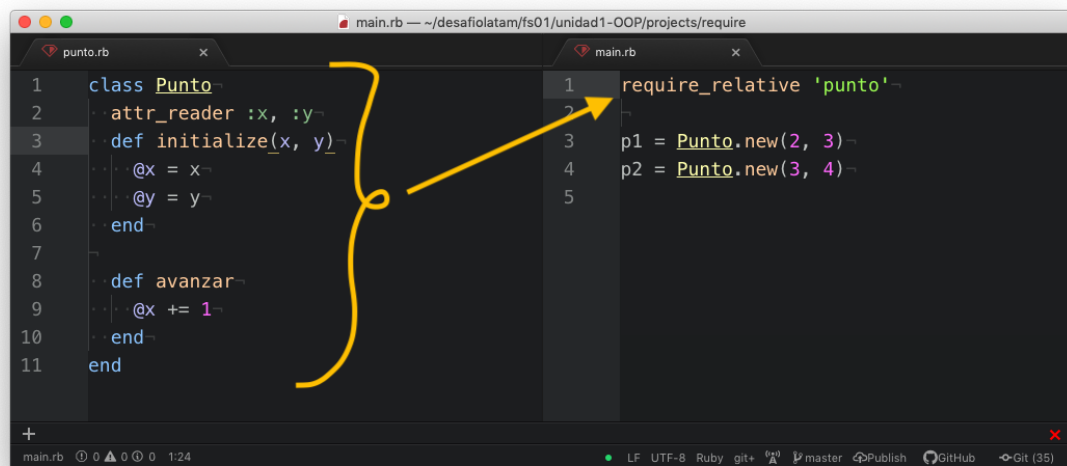
Introducción

La programación orientada a objetos busca la reutilización de código, pero esto es muy difícil de lograr si todo el código está en un solo archivo. En este capítulo aprenderemos el proceso para cargar nuestros objetos desde archivos distintos.

Para cargar el código desde otro archivo utilizaremos el método `require_relative`

La instrucción `require_relative`

El método `require_relative` nos permite cargar un archivo dentro de nuestro código, de esta forma podemos ocupar todas las definiciones que se encuentre dentro del archivo.



Como argumento recibe el nombre del archivo y asume la extensión `.rb` pero también podemos especificarla.

Trabajaremos con el ejercicio de los puntos

Para realizar nuestro primer ejercicio de separación utilizaremos el ejercicio de los puntos que desarrollamos en el capítulo anterior.

```

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

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

```

Nuestro objetivo es separar la clase **Punto** del uso de la clase. A continuación realizaremos el proceso desde cero para aprender el procedimiento.

Separando los archivos

El primer paso para facilitar la reutilización es dejar cada clase en un archivo distinto y un archivo principal que carga estos objetos.

Por lo que hay que crear un archivo `punto.rb` para guardar el siguiente código:

```

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

```

Se debe crear otro archivo llamado `main.rb` donde se agrega lo siguiente:

```

require_relative punto # require_relative nos permite cargar archivos
p1 = Punto.new(2,3)
p1.avanzar

```

Cargando el proyecto

Para cargar el proyecto simplemente hay que escribir en el terminal

```

ruby main.rb

```

Utilizando una clase desde IRB

En algunas ocasiones, puede que se necesite ocupar una clase definida dentro de uno de los archivos en IRB. Parase repetirá el proceso de utilizar `require_relative`

Al entrar a IRB desde la misma carpeta donde se creó el archivo `punto.rb` se tiene que escribir el siguiente código `require_relative 'punto'` para obtener como resultado `true` o `false`, dependiendo de si es primera vez que cargamos el archivo o no.

Cuidado

`require_relative` no carga dos veces el mismo archivo por lo que si modificamos nuestra clase tendremos que salir de IRB y entrar nuevamente.

La instrucción require y require_relative

Existe otra instrucción llamada `require`, ambas permiten cargar otros archivo dentro del código.

Utilizaremos `require` cuando los archivos a cargar estén definidos dentro del path como es el caso de las gemas y utilizaremos `require_relative` cuando estemos cargando nuestros propias clases.

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

Introducción

Cuando se trabaja con objetos es muy frecuente trabajar con más de uno, por ejemplo, una persona puede tener una mascota, o un arreglo puede contener otros objetos.

En esta sección se va a trabajar con objetos que se pueden asociar. El tipo de asociación a estudiar recibe el nombre de Asociación fuerte.



Cardinalidad y dirección

Al igual que en los diagramas de bases de datos, en las asociaciones de un diagrama UML se debe indicar la cardinalidad. En este diagrama se ve que una Persona puede tener 0 o 1 mascota, la dirección de la flecha indica la navegación posible, o sea este diagrama indica de que de una persona se puede llegar a una mascota pero de una mascota no se puede llegar a una persona.

Traspassando el diagrama a código

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

class Mascota
  attr_accessor :nombre
end
```

Para probar el código anterior a, se debe agregar los constructores que permitan asignar un nombre a ambas clases y mascota a las personas.

```
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", m1)
```

Revisando cardinalidad nula

Falta revisar la cardinalidad.

El diagrama anterior dice que la cardinalidad puede ser cero o uno, sin embargo, el método constructor de persona obliga a especificar una mascota.

Al crear un objeto sin mascota se obtiene el siguiente error:

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

Habilitando cardinalidad 0

Utilizando un argumento opcional se puede flexibilizar esto

```
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

Cuando la cardinalidad es N , se puede utilizar un arreglo, en lugar de simplemente asignar el atributo, se agregará al arreglo como se muestra en código siguiente:

```
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:0x00007f87e88c8210 @nombre="Nibbler">]
```

En el ejercicio anterior se utilizó una suposición, que el constructor inicial solo podía recibir una mascota, pero esto no necesariamente debe ser así, dependerá de los requerimientos específicos del sistema que siempre se quiere consultar.

Para contrastar es necesario repetir el ejercicio asumiendo lo contrario. El constructor de Persona debe recibir un arreglo de mascotas:

```
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:0x00007f87ea0d83c8 @nombre="Nibbler">
      # ]>
```

Si todo es un objeto ¿Por qué no todo es una asociación?

En Ruby casi todo es un objeto, los enteros, los strings y cualquiera de los otros datos que hasta ahora se tratan de una forma distinta. Usualmente se utilizan atributos para representar objetos pequeños, como números, fechas, strings y símbolos, mientras se reservan las asociaciones para objetos que se tienen que construir.

Nota: El consejo mencionado es propiedad de Martin Fowler, el creador de UML Distilled, un libro muy interesante sobre UML que es recomendado en caso que se desee profundizar en el tema.

Ejercicio de repaso

Una persona puede tener múltiples redes sociales, las personas tienen un nombre y una edad, mientras que de la información de la red social se requiere saber el tipo de red social, que puede ser Facebook, Instagram, Pinterest, Twitter o LinkedIn y además el usuario en la red social.

Solución

El primer paso es distinguir los atributos simples de aquellos objetos independientes. la clase usuario tiene nombre y edad, mientras que la clase red social tiene tipo y un nombre de usuario.

En cuanto a la cardinalidad, el enunciado menciona que puede tener múltiples redes sociales, información útil para comenzar a diagramar y luego generar el código:

```
class Persona
  attr_accessor :nombre, :edad, :redes_sociales
end

class RedSocial
  attr_accessor :tipo, :nombre_usuario
end
```

Para probar el código es necesario agregar los constructores, que permiten asignar valores.

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

class RedSocial
  attr_accessor :tipo, :nombre_usuario
  def initialize(tipo, nombre_usuario)
    @tipo = tipo
    @nombre_usuario = nombre_usuario
  end
end

# Probamos agregando datos

rs1 = RedSocial.new(:facebook, 'lop2034')
rs2 = RedSocial.new(:twitter, '@lop2034')
Persona.new('Fernando', 29, [rs1, rs2])

# => <Persona:0x00007f87e99cdbc8 @nombre="Fernando", @edad=29,
# @redes_sociales=[
#   <RedSocial:0x00007f87e99cdd08 @tipo=:facebook,
@nombre_usuario="lop2034">,
#   <RedSocial:0x00007f87e99cdc90 @tipo=:twitter, @nombre_usuario="@lop2034">
# ]>
```

```
#<Persona:0x00007fa1a9947b20 @nombre="Fernando", @edad=29, @redes_sociales=[#
<RedSocial:0x00007fa1a9947be8 @tipo=:facebook, @nombre_usuario="lop2034">, #
<RedSocial:0x00007fa1a9947b98 @tipo=:twitter, @nombre_usuario="@lop2034">]>
```


Ejercicio rectas:

Se solicita crear la clase recta, como es de conocimiento común, una recta está construida a partir de dos puntos.

```
class Recta
  def Initialize(p1, p2)
    @p1 = p1
    @p2 = p2
  end
end

Recta.new(Punto.new(2,3), Punto.new(3,4))
```

El constructor en este caso no cambia absolutamente nada, Ruby es un lenguaje de tipado dinámico, esto quiere decir que no importa que reciba un método mientras se comporte , o sea, mientras tenga los métodos necesarios.

Introducción a la identidad

Objetivos:

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

Introducción:

Así como las personas tienen una tarjeta de identificación, cada objeto tiene un identificador asociado para distinguirlo de otros objetos.



Este identificador permite saber si se está hablando de dos objetos distintos o del mismo objeto. En esta oportunidad, se busca que aprendas a ver este identificador y estudiar las ventajas de que cada objeto sea independiente de los otros.

¿Qué es identidad?

¿Qué hace único a un objeto? Si dos objetos tienen el mismo contenido ¿Son iguales?

El siguiente ejemplo nos aclarará la situación:

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

p1 = Persona.new("Trinidad")
p2 = Persona.new("Trinidad")
puts p1 == p2 # false
```

```
false
```

Aquí se observa que tanto persona1 como persona2 son personas distintas, independiente de que tengan el mismo nombre.

¿Cómo podemos probar que son distintas?

Si se modifica el nombre de cualquiera de las personas, la otra no se verá afectada.

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

```
Trinidad
```

Son instancias distintas por que al modificar una no modificamos la otra

Otro ejemplo similar:

- Construir la clase persona, con nombre y la cantidad de km caminados.
- Toda persona parte con 0 km caminados.
- Instanciar dos personas a partir de esta clase.

```
class Persona
  def initialize(nombre, caminado = 0)
    @nombre = nombre
    @caminado = caminado
  end

  def caminar(km = 1)
    @caminado += km
  end

  def caminado
    @caminado
  end
end

p1 = Persona.new("Javiera")
p1.caminar(5)
p1.caminar

p2 = Persona.new("Javiera")
p2.caminar(10)

puts p1.caminado
puts p2.caminado
```

```
6
10
```

A partir de este código se desprende algunas observaciones interesantes:

- Instanciar dos personas, donde ambas se llaman *Javiera*. Sin embargo, no son la misma persona.
- Los kilómetros caminados por cada una de estas personas son distintos, no se suman entre ellas, no se confunde la información que pertenece a cada una, esto se debe nuevamente a que **son dos personas distintas** y formalmente se debe decir que: **son dos instancias distintas**.

¿Cómo obtener el identificador de un objeto?

Toda instancia tiene un identificador único, se puede saber cual es, utilizando el método `.object_id`

```
class Persona
end

p1 = Persona.new
p2 = Persona.new
puts p1.object_id #70351254781520
puts p2.object_id #70351254781500
```

Cada vez que se crea una instancia nueva, este instancia recibe un identificador. El identificador permite saber si se está hablando de dos objetos distintos o del mismo objeto, independiente del contenido que tengan cosa que sucede con todo tipo de objetos.

Probando `.object_id` con los arrays

```
puts [1,2,3,4].object_id # 70110849102780
puts [1,2,3,4].object_id # 70110858362060
```

```
70166147446980
70166147444600
```

Probando `.object_id` con los strings

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

```
70166159749440
70166159747080
```

Las excepciones

Probando `.object_id` con enteros

```
2.object_id # 5  
2.object_id # 5
```

Algunas instancias tienen siempre el mismo id, esto sucede en casos muy particulares como integers, floats, símbolos, true y false.

Probando `object_id` con símbolos

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

Por ahora solo

Resumen:

En este capítulo introdujimos el concepto de identidad de un objeto. Aprendimos que dos objetos que tienen el mismo contenido no implica que sean exactamente el mismo objeto y que podemos ocupar el método `.object_id` para conocer el identificador de un objeto.

En el próximo capítulo estudiaremos las implicancias de la identidad y los conceptos de mutabilidad e inmutabilidad.

Identidad y variables

Objetivos

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

Introducción

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

En el ejemplo anterior, todos los casos que se estudiaron se asignaron objetos distintos a variables, pero en algunos casos se puede tener el mismo objeto en dos variables distintas.

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

```
true
```

En estas situaciones se debe tener cuidado porque al modificar uno, se modifica el otro automáticamente

```
# Si trabajamos con el mismo objeto.

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

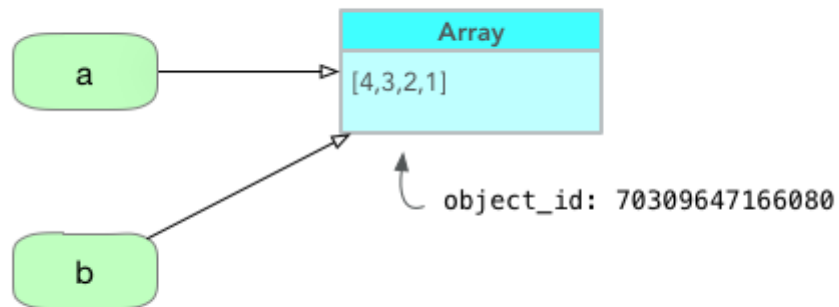
# En la variable b guardamos a
puts a.object_id == b.object_id # true

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

# Al cambiar a, cambiamos b
```

```
true
[8, 2, 3, 4]
```

Ilustrando lo sucedido



Como vemos en el siguiente gráfico, tanto la variable a como la variable b contienen el mismo objeto, y por lo tanto modificar el contenido de una, modifica el contenido de la otra.

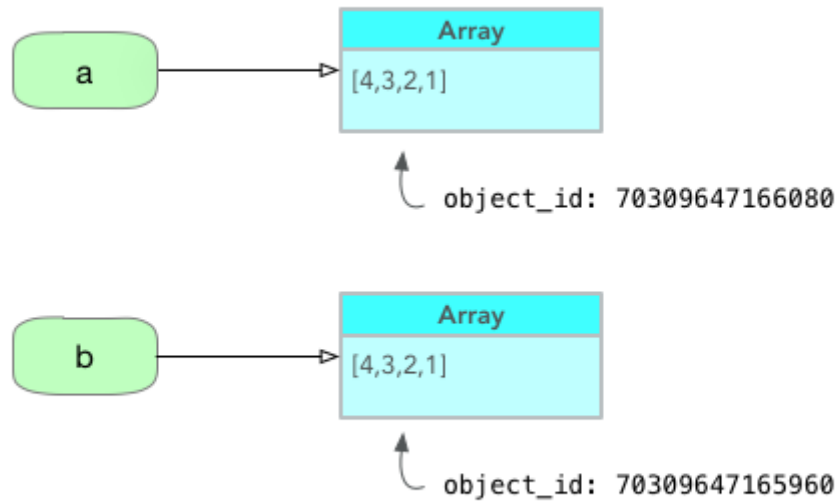
Esto no sucede si se trabaja con objetos distintos

```
# Si trabajamos con un objeto distinto (aunque tenga el mismo contenido)

a = [1, 2, 3, 4]
b = [1, 2, 3, 4] # instanciamos un nuevo objeto

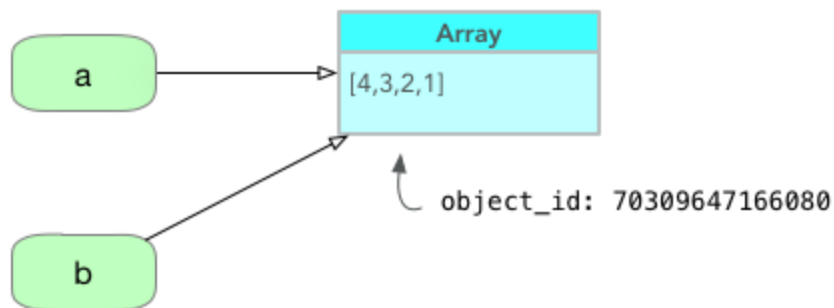
puts a.object_id == b.object_id # false
a[0] = 8
print b # [1, 2, 3, 4]

# al cambiar a NO cambia b, son objetos distintos.
```

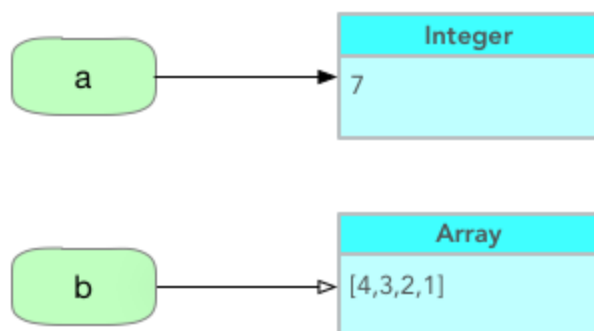


Asignar un nuevo valor a una variable no es lo mismo que modificar el objeto

Cuando se asigna un objeto nuevo a una variable, debe cambiarse la referencia.



```
a = 7
```



Si se hubiese asignado un array a la variable `a` habría sucedido lo mismo, aquí lo importante es que cambiar el contenido de un objeto es distinto a asignar un nuevo objeto a una variable.

Modificar V.S Asignar

Por definición se modifica el contenido cuando se cambia algunas de las propiedades del objeto.

Ejemplo:

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

en este caso se asigna un nuevo valor cuando se instancia un nuevo objeto y se guarda en la variable

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

Mutabilidad

- 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

En este capítulo se formalizará lo aprendido anteriormente introduciendo el concepto de 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.

Anteriormente se dijo que un objeto es mutable si puede cambiar de estado, pero, ¿Qué es cambiar de estado?

Un objeto cambia de estado cuando se modifica uno de sus atributos, para probar esto se debe contruir una clase base llamada MoldeAuto

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

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

Agregando un setter

En caso de querer cambiar el color , aparece el siguiente problema, no se puede modificar directamente un estado de un objeto, si no que hay que hacerlo a través de un método. Para lograrlo hay que tener 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
```

```
"rojo"
```

Debido a que m1 puede cambiar de estado, se dice que es un objeto mutable, y por lo mismo hay que tener las consideraciones necesarias que se han estudiado hasta ahora.

```
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

Un objeto está compuesto de métodos y atributos, cuando sus atributos pueden cambiar debido al llamado de un método entonces, se puede inferir que el objeto es mutable.

Casi todo objeto puede llegar a tener un método que modifique algún atributo, por lo mismo mas que preocuparse de si el objeto es mutable o no, se debe poner atención en los métodos que se llamarán y los efectos que estos puedan tener.

El caso de los arrays

Methods

```
::[]  
::new  
::try_convert  
#&  
#*  
#+  
#-  
#<<  
#<=>  
#==  
#[]  
#[]=  
#abbrev  
#assoc  
#at  
#bsearch  
#clear  
#collect  
#collect!  
#combination  
#compact  
#compact!  
#concat  
#count  
#cycle  
#dclone  
#delete  
#delete_at  
#delete_if  
#drop  
#drop_while  
#each  
#each_index  
#empty?  
#eq?  
#fetch  
#fill  
#find_index  
#first  
#flatten  
#flatten!
```

Son un caso muy interesante, ya que tienen distintas versiones de métodos, los que modifican el objeto están marcados con un signo de exclamación, mientras que los que devuelven un objeto nuevo no lo tienen.

Creando un método que modifique el estado

Cualquier método que modifique directamente un estado de un objeto es un método mutable.

```
class Persona
  def initialize(nombre, caminado = 0)
    @nombre = nombre
    @caminado = caminado
  end

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

  def caminado
    @caminado
  end
end

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

En este caso caminar es un método mutable porque afecta directamente sobre el estado **caminado** de la persona

Creando un método que no modifique el estado

Aquí viene una pregunta interesante, ¿Cómo se puede crear un método que realice un cambio en el objeto, pero a la vez no lo realice?.

El truco es crear un objeto nuevo con el estado modificado pero sin modificar el original.

```
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>
```

Cuando se llama a un método inmutable , siempre como resultado se tiene que se ha creado un nuevo objeto. A simple vista puede verse como algo extraño pero ya se ha ocupado , probablemente sin saberlo.

Por ejemplo cuando se concatenan dos strings.

```
a = "hola"
b = " mundo"
c = a + b
puts a.object_id
puts b.object_id
puts c.object_id
```

Al llamar al método `+` del string "hola" se genera un objeto nuevo sin modificar el anterior.

Recordemos que `+` es un método

Dentro de ruby los operadores son métodos de instancia de una clase. Esto permite crear abstracciones potentes.

Imaginemos que tenemos una clase cuadrado de lego, y tenemos dos piezas y las juntamos, eso sería una pieza nueva del tamaño de las dos anteriores.

```
class Lego
  attr_reader :size
  def initialize(size = 1)
    @size = size
  end
  def +(lego)
    Lego.new(@size + lego.size)
  end
end

Lego.new(2) + Lego.new()
```

```
#<Lego:0x00007fa1a89adde0 @size=3>
```

Ejercicio resuelto

Modifica el código de la clase Lego para que la suma modifique el tamaño del objeto actual en lugar de devolver uno nuevo.

```
class Lego
  attr_reader :size
  def initialize(size = 1)
    @size = size
  end
  def +(lego)
    @size += lego.size
  end
end

Lego.new(2) + Lego.new()
```

3

¿Modificar o no modificar?

La ventaja de los métodos que no modifican el estado de un objeto es que difícilmente que puedan afectar nuestro código por descuido. La ventaja de los que si los modifican es que no requieren de la creación de un objeto nuevo.

Ejemplo de canastas de regalo

Otro ejemplo, suponga que se 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.

```
class Canasta
  attr_accessor :frutas, :velas, :tarjetas
  def initialize(frutas, velas, tarjetas)
    @frutas = frutas
    @velas = velas
    @tarjetas = tarjetas
  end
  def +(otra_canasta)
    Canasta.new(@frutas + otra_canasta.frutas,
                @velas + otra_canasta.velas,
                @tarjetas + otra_canasta.tarjetas)
  end
  def cantidad_de_elementos
    @frutas + @tarjetas + @velas
  end
end
```

```
end
end

fusion_canastas = Canasta.new(2, 1, 3) + Canasta.new(5, 2, 3)
puts fusion_canastas.cantidad_de_elementos
```

16

Decisiones de diseño

¿Por qué hacer la suma de canasta inmutables?

Tiene sentido que al juntarse dos canastas se sumen todos los elementos sobre la primera? O ¿Tiene más sentido que se cree una canasta nueva?

Estas decisiones de diseño se tienen que hacer de forma frecuente. Ambas opciones son válidas, lo importante es ser coherente a lo largo del código, y documentar bien estos procesos.

Por ejemplo todas las operaciones mutables sobre los arrays de ruby llevan el signo de exclamación al final y esto hace sencillo recordar que hace cada una, ser consistente y evitar errores.

Ejercicio desarrollado

Desarrollar la clase mascota pero esta vez el método cruzar devolverá un objeto nuevo.

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

  def cruzar(otra_mascota)
    Mascota.new("Mascota nueva")
  end
end

# Ejemplo de uso
m1 = Mascota.new("Fluffy") # => <Mascota:0x007fefb2120f58>
m2 = Mascota.new("Laika") # => #<Mascota:0x007fefb20dd118>
hijo = m1.cruzar(m2) # => <Mascota:0x007fefb2087790>
```


Manejo de excepciones

Objetivos:

- Levantar excepciones
- Manejar excepciones levantadas

Introducción

Una excepción es el punto de quiebre de un programa, todo va bien hasta que el programa se encuentra con un problema, en este momento lo que hace es levantar una excepción e interrumpe su proceso si esa excepción no es manejada.

Programa funciona -> Se encuentra un problema -> Se levanta excepción -> Se intenta rescatar excepción -> Fin

Ejemplo de excepción

Dentro de nuestro código en algún momento podríamos intentar sumar un string con un entero.

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

Al suceder esto el programa se detendría y nos mostraría la excepción y el traceback.

Manejando las excepciones

Las excepciones no son el punto final de un programa, se pueden manejar, todas las excepciones que ocurran dentro de un bloque begin se pueden rescatar con un rescue.

```
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'
```

El antipatron begin-end

Uno podría tentarse a envolver todo el código en un `begin` y un `end` pero es una pésima idea, las excepciones ayudan a encontrar errores en el código y sirven para indicar a los programadores lo que están haciendo algo mal.

Por lo mismo, a la hora de programar podemos levantar excepciones para mostrarles a otros programadores que están ocupando mal nuestros objetos.

Levantando una excepción

Para levantar una excepción manualmente se puede ocupar la instrucción `raise`

```
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

Esta excepción también se puede rescatar utilizando la instrucción `rescue`

```
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'
```

En pantalla se observa: Fui salvado aunque detecté el problema: Error tipo 409 y ahora el programa corre de forma normal

Introducción a tipos de Excepciones

Existen diversos tipos de excepciones, cuando se utiliza `raise` se levanta una excepción del tipo `RuntimeError`. que vendría siendo simplemente un "error en tiempo de ejecución".

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

```
RuntimeError
```

ArgumentError

Hay excepciones mas específicas, una que se ocupa bastante es ArgumentError, por lo que es conveniente ver un ejemplo donde es útil. se quiere hacer un método que sume dos valores, pero se quiere asegurarse que sean enteros.

```
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
```

Es justo en este ejemplo donde se empieza a entender la importancia de las excepciones. Se puede ocupar excepciones en los objetos para indicarle a otros programadores o a nosotros mismos que los estamos ocupando y/o usando mal.

Mejorando un ejercicio

En un ejemplo anterior se creó la clase Persona y la clase Mascota en donde una Persona podía tener una mascota.

Si se retoma este ejercicio asegurando que no se puede pasar un objeto distinto a uno del tipo mascota, para evitar errores con el código, se vuelve a agregar el código base :

```
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
```

```
:initialize
```

Ahora se necesita evitar que la variable mascota sea algo distinto a la clase mascota, para eso se debe ocupar la excepción llamada ArgumentError

```

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

```

Resumen

En ruby existen las excepciones, estas indican errores en los programas y detienen los programas, pero también existen instrucciones como `rescue` que permiten continuar a pesar de algún problema.

Existen distintos tipos de excepciones pero se mencionan solo dos, `RuntimeError` el cual es un error genérico y `ArgumentError` el cual es un tipo de error utilizado cuando los argumentos que recibe un método son incorrectos.

La lección importante de este capítulo consiste en que las excepciones pueden ser utilizadas para indicarle a otros programadores que están ocupando nuestro código de forma equivocada, y esto es especialmente útil ya que se trabaja con objetos y el propósito de estos es reutilizar código.

Rescatando una excepción específica

Se debe tener cuidado de no caer en el antipatrón de meter todo el código dentro de un `rescue` y se debe tener cuidado de no rescatar excepciones genéricas, porque es posible que se esté ocultando otro error.

Cuando rescatemos excepciones debemos ser específicos respecto a la excepción.

```

begin
  # -
rescue ArgumentError
  # -
end

```

Sin embargo no profundizaremos en los casos de uso de rescate, volveremos a verlo cuando trabajemos con transacciones en bases de datos. Ahí tendremos la oportunidad de enfrentar este tema con casos prácticos.

Cierre

En esta unidad se trabajaron diversos aspectos críticos de la programación orientada a objetos que no se deben perder de vista:

1. El propósito de la programación orientada a objetos es facilitar la reutilización de código
2. Para lograr la reutilización estudiamos dos principios claves: La abstracción y la encapsulación. Existen dos mas que estudiaremos en la próxima unidad
3. La abstracción nos dice que debemos pensar en los objetos como cajas negras, cuando utilicemos los objetos nos importa lo que hacen y no el como lo hacen. Obviamente cuando programamos los objetos tenemos que definir el como.
4. La encapsulación nos dice que el contenido de estas cajas negras no puede ser modificado directamente, tenemos que crear métodos para obtener y modificar los valores, estos son getters y setters pero también podemos crear otros métodos que modifiquen el valor antes de devolverlo o que cambien a través de una fórmula el valor.
5. Los objetos en Ruby son mutables, esto quiere decir que su contenido se puede modificar, sin embargo si necesitamos que sean inalterables nosotros podemos crear métodos que devuelvan objetos nuevos sin necesidad de cambiar los atributos existentes.
6. Ruby es un lenguaje de tipado dinámico, esto quiere decir que los métodos no revisan el tipo de dato al momento de ser llamado, pero podemos ocupar excepciones para asegurarnos que los métodos reciban exactamente el tipo que necesitamos.