



# Creando a Pepita

- 

Inicialmente en el ambiente solo existen objetos simples como números, strings y booleanos.

Pero como es imposible que quienes diseñan un lenguaje puedan precargar objetos para solucionar todos nuestros problemas, también nos dan la posibilidad de crear los nuestros.

En Ruby, si quisiéramos **definir** a `Norita`, escribiríamos el siguiente código:

```
module Norita  
end
```

Sí, así de simple.

¿Te animás a modificar nuestro código para crear a `Pepita`?

 Solución  Consola

```
1 module Pepita  
2 end
```

 Enviar

 ¡Muy bien! Tu solución pasó todas las pruebas

¡Muy bien, [Pepita](#) vive!

Como dedujiste, la **definición** de un objeto se inicia con la palabra reservada [module](#) , luego el nombre del objeto (con la primera letra en mayúscula) y su fin se indica con un [end](#) .

---

Esta guía fue desarrollada por Federico Aloï, Franco Bulgarelli, Ariel Umansky bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# Pepita, ¿me entendés?

•

En la lección anterior `Pepita` entendía los mensajes `comer_lombriz!`, `cantar!`, `volar_en_circulos!` y `energia`.

Con la definición que construimos recién, ¿podrá responderlos?

Intentá enviarle a `Pepita` los mensajes habituales y fijate qué sucede.

💡 ¡Dame una pista!

[➤ Consola](#)

[⌕ Biblioteca](#)

>



Esta guía fue desarrollada por Federico Aloí, Franco Bulgarelli, Ariel Umansky bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)







# Los mejores, los únicos, los métodos en objetos

¿Otra vez `undefined method`? ¿Y ahora qué falta?

Para que un objeto entienda un mensaje debemos "enseñarle" cómo hacerlo, y para ello es necesario definir un **método** dentro de ese objeto:

```
module Pepita
  def self.cantar!
  end
end
```

Un método es, entonces, la descripción de **qué hacer cuando se recibe un mensaje del mismo nombre**.

Dos cosas muy importantes a tener en cuenta :

- Todos los métodos **comienzan con `def` y terminan con `end`**. Si nos falta alguna de estos dos la computadora no va a entender nuestra solución.
- Todos los métodos que pertenezcan al mismo objeto van **dentro del mismo `module`**.

Agregale a la definición de `Pepita` los métodos necesarios para que pueda responder a los mensajes `cantar!`, `comer_lombriz!` y `volar_en_circulos!`.

💡 ¡Dame una pista!

 Solución  Consola

```
1 module Pepita
2   def self.cantar!
3   end
4   def self.comer_lombriz!
5   end
6   def self.volar_en_circulos!
```



```
7 end
8 end
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Perfecto, ahora `Pepita` entiende casi todos los mismos mensajes que en la lección anterior. Pero, ¿hacen lo mismo?

Antes de seguir, enviá algunos de los mensajes en la **Consola** y fijate qué **efecto** producen sobre nuestra golondrina.

---

Esta guía fue desarrollada por Federico Aloí, Franco Bulgarelli, Ariel Umansky bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# Perdiendo energía



Acabamos de aprender una de las reglas fundamentales del envío de mensajes: si a un objeto no le decimos **cómo** reaccionar ante un mensaje, y se lo envíamos, no lo entenderá y nuestro programa se romperá. Y la forma de hacer esto es **definiendo un método**.

Ahora bien, los métodos que definiste recién no eran muy interesantes: se trataba de *métodos vacíos* que evitaban que el programa se rompiera, pero no hacían nada. En realidad, **Pepita** tiene energía y los diferentes mensajes que entiende deberían modificarla.

¿Cómo podríamos decir que cuando **Pepita** vuela, pierde **10** unidades de energía? ¿Y que inicialmente esta energía es **100**? Así:

```
module Pepita
  @energia = 100

  def self.volar_en_circulos!
    @energia = @energia - 10
  end
end
```

Una vez más, ya definimos a **Pepita** por vos. Probá, en orden, las siguientes consultas:

```
> Pepita.volar_en_circulos!
> Pepita.volar_en_circulos!
> Pepita.energia
```

Puede que los resultados te sorprendan, en breve hablaremos de esto.

>



---

Esta guía fue desarrollada por Federico Aloj, Franco Bulgarelli, Ariel Umansky bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)







# Atributos

Analicemos el código que acabamos de escribir:

```
module Pepita
  @energia = 100

  def self.volar_en_circuitos!
    @energia = @energia - 10
  end
end
```

Decimos que `Pepita` *conoce* o *tiene* un nivel de energía, que es variable, e inicialmente toma el valor `100`. La energía es un **atributo** de nuestro objeto, y la forma de **asignarle** un valor es escribiendo `@energia = 100`.

Por otro lado, cuando `Pepita` recibe el mensaje `volar_en_circuitos!`, su energía disminuye: se realiza una nueva **asignación** del atributo y pasa a valer lo que valía antes (o sea, `@energia`), menos `10`.

Como la operación `@energia = @energia - 10` es tan común, se puede escribir `@energia -= 10`. Como te imaginarás, también se puede hacer con la suma.

Sabiendo esto:

- cambiá la definición del método `volar_en_circuitos!` para que utilice la expresión simplificada;
- definí la versión correcta del método `comer_lombriz!`, que provoca que `Pepita` gane `20` puntos de energía;

 Solución  Consola

```
1 module Pepita
2   @energia = 100
3
4   def self.volar_en_circuitos!
```

```
5      @energia -= 10
6  end
7
8  def self.comer_lombriz!
9      @energia += 20
10 end
11 end
```



▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Acabamos de aprender un nuevo elemento del paradigma de objetos: los **atributos** (los cuales escribiremos anteponiendo `@`), son objetos que nos permiten representar una característica de otro objeto. Un objeto conoce a todos sus atributos por lo que puede enviarles mensajes, tal como hicimos con `@energia`.

*Entonces, si le pude enviar mensajes a `@energia`, ¿eso significa que los números también son objetos?*

¡Claro que sí! ¡Todo-todo-todo es un objeto!

Esta guía fue desarrollada por Federico Aloï, Franco Bulgarelli, Ariel Umansky bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# Conociendo el país

Hasta ahora los métodos que vimos solo producían un efecto. Si bien solo pueden devolver una cosa, pueden producir varios efectos!

Solo tenés que poner uno debajo del otro de la siguiente forma:

```
def self.comprar_libro!  
  @plata -= 300  
  @libros += 1  
end
```

Como te dijimos, **Pepita** podía volar a diferentes ciudades. Y cuando lo hace, cambia su ciudad actual, además de perder 100 unidades de energía. Las distintas ciudades vas a poder verlas en la **Biblioteca**.

Con esto en mente:

- Creá un atributo **ciudad** en **Pepita**: la ciudad donde actualmente está nuestra golondrina.
- Hacé que la **ciudad** inicial de pepita sea **Iruya**.
- Definí un método **volar\_hacia!** en **Pepita**, que tome como argumento otra ciudad y haga lo necesario.

💡 ¡Dame una pista!

**Solución** **Biblioteca** **Consola**

```
1 module Pepita  
2   @energia = 100  
3   @ciudad = Iruya  
4  
5   def self.volar_en_circulos!  
6     @energia -= 10  
7   end  
8  
9   def self.comer_lombriz!  
10    @energia += 20  
11  end  
12  
13  def self.volar_hacia! lugar  
14    @energia -= 100  
15    @ciudad = lugar  
16  end  
17 end
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Esta guía fue desarrollada por Federico Aloí, Franco Bulgarelli, Ariel Umansky bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# Leyendo el estado

Antes te mostramos que si enviamos el mensaje `energia`, fallará:

7. Leyendo el estado

```
> Pepita.energia
undefined method `energia' for Pepita:Module (NoMethodError)
```

El motivo es simple: **los atributos NO son mensajes**.

Entonces, ¿cómo podríamos consultar la energía de `Pepita`? Definiendo un método, ¡por supuesto!

```
module Pepita
  #...atributos y métodos anteriores...

  def energia
    @energia
  end
end
```

Ya agregamos el método `energia` por vos. Probá en la consola ahora las siguientes consultas:

```
> Pepita.energia
> Pepita.energia = 120
> energia
```

¿Todas las consultas funcionan? ¿Por qué?

>



Esta guía fue desarrollada por Federico Aloí, Franco Bulgarelli, Ariel Umansky bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)


[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# Cuestión de estado

Los objetos pueden tener múltiples atributos y al conjunto de estos atributos se lo denomina **estado**. Por ejemplo, si miramos a [Pepita](#)  8. Cuestión de estado

```
module Pepita
  @energia = 100
  @ciudad = Obera

  #...etc...
end
```

Lo que podemos observar es que su estado está conformado por `ciudad` y `energia`, dado que son sus atributos.

El estado es siempre **privado**, es decir, solo el objeto puede utilizar sus atributos, lo que explica por qué las siguientes consultas que hicimos antes fallaban:

```
> Pepita.energia = 100
> energia
```

Veamos si se entiende: mirá los objetos en la solapa **Biblioteca** y escribí el estado de cada uno.

 **Solución**  Biblioteca  Consola

```
1 estado_pepita = %w(
2   energia
3   ciudad
4 )
5
6 estado_kiano1100 = %w(
7 )
8
9 estado_rolamotoC115 = %w(
10 )
11
12 estado_enrique = %w(
13   celular
14   dinero_en_billetera
15   frase_favorita
16 )
```

 Enviar

 ¡Muy bien! Tu solución pasó todas las pruebas

Esta guía fue desarrollada por Federico Aloí, Franco Bulgarelli, Ariel Umansky bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)







## ¿Dónde estás?

Queremos saber dónde se encuentra `Pepita`, para lo cual necesitamos agregarle un mensaje `ciudad` que nos permita **acceder** al atributo del mismo nombre.

Inspirándote en la definición de `energia`, definí el método `ciudad` que retorne la ubicación de nuestra golondrina.

Solución Consola

```
1 module Pepita
2   @energia = 100
3   @ciudad = Obera
4
5   def self.energia
6     @energia
7   end
8
9   def self.ciudad
10    @ciudad
11  end
12
13  def self.cantar!
14    'pri pri pri'
15  end
16
17  def self.comer_lombriz!
18    @energia += 20
19  end
20
21  def self.volar_en_circulos!
22    @energia -= 10
23  end
24
25  def self.volar_hacia!(destino)
26    @energia -= 100
27    @ciudad = destino
28  end
29 end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

A estos métodos que sirven para conocer el valor de un atributo los llamamos **métodos de acceso** o simplemente **accessors**, por su nombre en inglés.

Esta guía fue desarrollada por Federico Aloí, Franco Bulgarelli, Ariel Umansky bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)



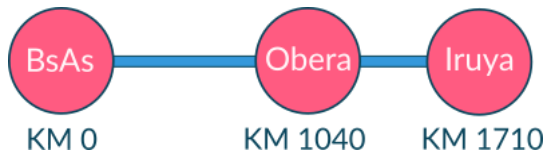




# Volando alto

Volar hacia un cierto punto no es tarea tan fácil: en realidad, **Pepita** pierde tanta energía como la mitad de kilómetros que tenga que recorrer.

Aunque en el mapa real no sea así, imaginaremos que las ciudades están ubicadas en línea recta, para facilitar los cálculos:



Por ejemplo, si **Pepita** está en **Obera** y quiere volar a **Iruya** debe recorrer 670 kilómetros, por lo que perderá 335 unidades de energía.

¿Y si **Pepita** está en **Iruya** y quiere volar a **Obera**? ¡También! La distancia entre dos ciudades siempre es un valor positivo. Para resolver esto contamos con el mensaje `abs` que entienden los números y nos retorna su valor absoluto:

```
> 17.abs
=> 17

> (-17).abs
=> 17

> (1710 - 1040).abs
=> 670

> (1040 - 1710).abs
=> 670

> (1040 - 1710).abs / 2
=> 335
```

Sabiendo esto:

- Definí el objeto que representa a **BuenosAires**.
- Definí en **Obera**, **Iruya** y **BuenosAires** un método `kilometro` que retorne la altura a la que se encuentran, según el esquema. ¡Cuidado! No tenés que guardar el valor en un atributo `@kilometro` sino simplemente retornar el número que corresponde.
- Modificá el método `volar_hacia!` de **Pepita** para hacer el cálculo de la distancia y alterar su energía. Para acceder al kilometro inicial de **Pepita** tenes que hacer `@ciudad.kilometro`.

Para que el ejemplo tenga sentido, vamos a hacer que **Pepita** arranque con la energía en 1000.

💡 ¡Dame una pista!

[Solución](#) [Consola](#)

```
1 %w(Definicion objseto BuenoAires con atributo kilometro y su valor en 0)
2 module BuenosAires
3   def self.kilometro
4     0
5   end
6 end
7
8 module Obera
9   def self.kilometro
10    1040
11  end
12 end
13
14 module Iruya
15   def self.kilometro
```

```
16     1710
17   end
18 end
19
20 module Pepita
21   @energia = 1000
22   @ciudad = Obera
23
24   def self.energia
25     @energia
26   end
27
28   def self.ciudad
29     @ciudad
30   end
31
32   def self.cantar!
33     'pri pri pri'
34   end
35
36   def self.comer_lombriz!
37     @energia += 20
38   end
39
40   def self.volar_en_circulos!
41     @energia -= 10
42   end
43
44   def self.volar_hacia!(destino)
45     @energia -= (destino.kilometro-@ciudad.kilometro).abs/2
46     @ciudad = destino
47   end
48 end
```

 Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

¡Buen trabajo!

Cuando programamos en este paradigma solemos tener a disposición un montón de objetos que interactúan entre sí, y por lo tanto aprender cuándo usarlos y definirlos es una habilidad fundamental, que irás adquiriendo con la práctica.

Esta guía fue desarrollada por Federico Aloï, Franco Bulgarelli, Ariel Umansky bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# Delegar es bueno

En el ejercicio anterior vimos que un objeto (en ese caso, [Pepita](#)) le puede enviar mensajes a otro que conozca (en ese caso, ciudades como [Obera](#) o [BuenosAires](#)):

```
module Pepita
  # ...etc...

  def self.volar_hacia!(destino)
    @energia -= (@ciudad.kilometro - destino.kilometro).abs / 2
    @ciudad = destino
  end
end
```

Esto se conoce como *delegar una responsabilidad*, o simplemente, **delegar**: la responsabilidad de saber en qué kilómetro se encuentra es de la ciudad, y no de [Pepita](#).

A veces nos va a pasar que un objeto tiene un método muy complejo, y nos gustaría subdividirlo en problemas más chicos que **el mismo objeto** puede resolver. Pero, ¿cómo se envía un objeto mensajes a sí mismo?

Un objeto puede enviarse un mensaje a sí mismo fácilmente usando `self` como receptor del mensaje.

```
module Pepita
  # ...etc...

  def self.volar_hacia!(destino)
    self.gastar_energia! destino #¡Ojo! No hicimos Pepita.gastar_energia!(destino)
    @ciudad = destino
  end

  def self.gastar_energia!(destino)
    @energia -= (@ciudad.kilometro - destino.kilometro).abs / 2
  end
end
```

Pero esto se puede mejorar un poco más. Delegá el cálculo de la distancia en un método `distancia_a`, que tome un destino y devuelva la distancia desde la ciudad actual hasta el destino.

[Solución](#) [Biblioteca](#) [Consola](#)

```
1 module Pepita
2   @energia = 1000
3   @ciudad = Obera
4
5   def self.energia
6     @energia
7   end
8
9   def self.ciudad
10    @ciudad
11  end
12
13  def self.cantar!
14    'pri pri pri'
15  end
16
17  def self.comer_lombriz!
18    @energia += 20
19  end
20
21  def self.volar_en_circuitos!
```

```
22  @energia -= 10
23  end
24
25  def self.volar_hacia!(destino)
26    self.gastar_energia!(destino)
27    @ciudad = destino
28  end
29
30  def self.gastar_energia!(destino)
31    @energia -= (self.distancia_a (destino) ) / 2
32  end
33
34  def self.distancia_a(city)
35    (@ciudad.kilometro - city.kilometro).abs
36  end
37 end
```



✓ ¡Muy bien! Tu solución pasó todas las pruebas

La **delegación** es la forma que tenemos en objetos de **dividir en subtarear**: separar un problema grande en problemas más chicos para que nos resulte más sencillo resolverlo.

A diferencia de lenguajes sin objetos, aquí debemos pensar dos cosas:

1. cómo dividir la subtarea, lo cual nos llevará a **delegar** ese comportamiento en varios **métodos**;
2. qué objeto tendrá la **responsabilidad** de resolver esa tarea.

Esta guía fue desarrollada por Federico Aloí, Franco Bulgarelli, Ariel Umansky bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# ¿Es mi responsabilidad?

Hay un pequeño problema conceptual con la solución anterior: ¿por qué `Pepita`, una golondrina, es responsable de calcular la distancia entre dos ciudades?

Dicho de otra manera, ¿es *necesario* contar con una golondrina para poder calcular la distancia entre dos lugares? ¿Cual es el objeto más pequeño que podría saber hacer esto?

¿Lo pensaste? La respuesta es simple: ¡la misma ciudad! Por ejemplo, `BuenosAires` podría entender un mensaje `distancia_a`, que tome otra ciudad y devuelva la distancia entre ésta y sí misma.

Modificá la solución del ejercicio anterior para que sean las ciudades las que calculan las distancias. Pensá que no solo `Obera` debe tener este método, sino también `BuenosAires` e `Iruya`, para cuando tenga que volver.

💡 ¡Dame una pista!

 Solución  Consola

```
1 module Obera
2   def self.kilometro
3     1040
4   end
5
6   def self.distancia_a(destino)
7     (self.kilometro - destino.kilometro).abs
8   end
9
10
11 end
12
13 module Iruya
14   def self.kilometro
15     1710
16   end
```



```
17
18   def self.distancia_a(destino)
19     (self.kilometro - destino.kilometro).abs
20   end
21
22
23 end
24
25 module BuenosAires
26   def self.kilometro
27     0
28   end
29
30   def self.distancia_a(destino)
31     (self.kilometro - destino.kilometro).abs
32   end
33 end
34
35
36
37 module Pepita
38   @energia = 1000
39   @ciudad = Obera
40
41   def self.energia
42     @energia
43   end
44
45   def self.ciudad
46     @ciudad
47   end
48
49   def self.cantar!
50     'pri pri pri'
51   end
52
53   def self.comer_lombriz!
54     @energia += 20
55   end
56
57   def self.volar_en_circulos!
58     @energia -= 10
59   end
60
61   def self.volar_hacia!(destino)
62     self.gastar_energia!(destino)
63     @ciudad = destino
```

```
64 end
65
66 def self.gastar_energia!(destino)
67   @energia -= (@ciudad.distancia_a (destino) ) / 2
68 end
69
70
71 end
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Esta guía fue desarrollada por Federico Aloj, Franco Bulgarelli, Ariel Umansky bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)

