



¿Pepita está feliz?

•

¿Te acordás de `Pepita`? Bueno, aunque no lo creas, también cambia de estados de ánimo. En nuestro **modelo** de Pepita, vamos a representar simplemente dos estados posibles: cuando está débil y cuando está feliz.

¿Y cuándo ocurre eso?

- Pepita está *débil* si su energía es menor que 100.
- Pepita está *feliz* si su energía es mayor que 1000.

Completá los métodos `debil?` y `feliz?` de `Pepita`.

Como en esta lección no vamos a interactuar con las ciudades, hemos quitado todo lo relacionado a ellas de `Pepita`. Esto solo lo hacemos para que te sea más fácil escribir el código, no lo intentes en casa.

💡 ¡Dame una pista!

 Solución  Consola

```
1 module Pepita
2   @energia = 1000
3
4   def self.energia
5     @energia
6   end
7
8   def self.volar_en_circulos!
9     @energia -= 10
10  end
11
12  def self.comer_alpiste!(gramos)
13    @energia += gramos * 15
14  end
15
```



```
16 def self.debil?  
17   self.energia < 100  
18 end  
19  
20 def self.feliz?  
21   self.energia > 1000  
22 end  
23 end
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

En Ruby, es una **convención** que los mensajes que devuelven booleanos (o sea, verdadero o falso) terminen con un `?`.

Intentá respetarla cuando crees tus propios mensajes, acordate que uno de los objetivos del código es **comunicar** nuestras ideas a otras personas... y las convenciones, muchas veces, nos ayudan con esto.

Esta guía fue desarrollada por Federico Aloí y [muchas personas más](#), 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](#)





Reencuentro alternativo

•

Si llegaste hasta acá, ya deberías saber que en programación existe una herramienta llamada **alternativa condicional**.

En Ruby, como en muchos otros lenguajes, esto se escribe con la palabra reservada `if`. Por ejemplo:

```
module Jose
  def self.acomodar_habitacion!
    self.ordenar!
    if self.tiene_sabanas_sucias?
      self.cambiar_sabanas!
    end
    self.tender_la_cama!
  end
end
```

Sabiendo cómo se escribe la alternativa condicional en Ruby queremos que `Pepita`, además de recibir órdenes, tenga sus momentos para poder hacer lo que quiera.

Obviamente, qué quiere hacer en un momento dado depende de su estado de ánimo:

- Si está débil, come diez gramos de alpiste, para recuperarse.
- Si no lo está, no hace nada.

Hacé que `Pepita` entienda el mensaje `hacer_lo_que_quiera!` que se comporte como explicamos.

 Solución  Consola

```
1 module Pepita
2   @energia = 1000
3
4   def self.energia
5     @energia
6   end
```

```
7
8  def self.volar_en_circuitos!
9    @energia -= 10
10  end
11
12  def self.comer_alpiste!(gramos)
13    @energia += gramos * 15
14  end
15
16  def self.debil?
17    self.energia < 100
18  end
19
20  def self.feliz?
21    self.energia > 1000
22  end
23
24  def self.hacer_lo_que_quiera!
25    if self.debil?
26      self.comer_alpiste!(10)
27    end
28  end
29
30
31 end
32
33
```

 Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Como acabamos de ver, la alternativa condicional es como en otros lenguajes. La diferencia radica en su sintaxis, es decir, cómo la escribimos.

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





Repitamos qué pasa si no



Hay veces que con un `if` alcanza, pero otras queremos hacer algo si no se cumple una condición. Como ya te podrás imaginar, donde hay un `if`, cerca anda un `else`!

```
module Jardinero
  def self.cuidar!(planta)
    if planta.necesita_agua?
      3.times { self.regar! planta }
    else
      self.sacar_bichos! planta
    end
  end
end
```

¿Y ese `times` qué es?

Es un mensaje que entienden los números que sirve para ejecutar una porción de código varias veces. En este caso regaríamos 3 veces la planta recibida como argumento.

Ahora que conocimos la existencia de `times` y vimos cómo hacer `else ...`

Modificá la solución para que si Pepita no está débil vuele en círculos 3 veces.

 Solución  Consola

```
1 module Pepita
2   @energia = 1000
3
4   def self.energia
5     @energia
6   end
7
8   def self.volar_en_circulos!
9     @energia -= 10
10  end
11
12  def self.comer_alpiste!(gramos)
```



```
13   @energia += gramos * 15
14   end
15
16   def self.debil?
17     self.energia < 100
18   end
19
20   def self.feliz?
21     self.energia > 1000
22   end
23
24   def self.hacer_lo_que_quiera!
25     if self.debil?
26       self.comer_alpiste!(10)
27     else
28       3.times { self.volar_en_circulos!}
29     end
30   end
31
32
33 end
34
35
36
37
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Esta guía fue desarrollada por Federico Aloí y [muchas personas más](#), 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](#)





Voy a hacer, pero como yo quiero

Algunas veces vamos a tener condiciones anidadas. En otras palabras, un `if` dentro de un `if` o un `else`. Como en este ejemplo:

4. Voy a hacer, pero como yo quiero

```
module Docente
  def self.nota_conceptual(nota)
    if nota > 8
      "Sobresaliente"
    else
      if nota > 6
        "Satisfactoria"
      else
        "No satisfactoria"
      end
    end
  end
end
```

Ahora que vimos estas condiciones anidadas que poco tienen que ver con el nido de Pepita, vamos a conocer el comportamiento definitivo de Pepita cuando hace lo que quiere:

- Si está débil, come diez gramos de alpiste, para recuperarse.
- Si no está débil pero sí feliz, vuela en círculos cinco veces.
- Si no está feliz ni débil, vuela en círculos 3 veces.

Modificá a `Pepita` para que el método `hacer_lo_que_quiera!` se comporte como mencionamos más arriba.

 Solución  Consola

```
1 module Pepita
2   @energia = 1000
3
4   def self.energia
5     @energia
6   end
```

```
7
8  def self.volar_en_circulos!
9    @energia -= 10
10 end
11
12 def self.comer_alpiste!(gramos)
13   @energia += gramos * 15
14 end
15
16 def self.debil?
17   self.energia < 100
18 end
19
20 def self.feliz?
21   self.energia > 1000
22 end
23
24 def self.hacer_lo_que_quiera!
25   if self.debil?
26     self.comer_alpiste!(10)
27   elsif self.feliz?
28     5.times { self.volar_en_circulos!}
29   else
30     3.times { self.volar_en_circulos!}
31   end
32 end
33
34
35 end
36
37
38
39
40
41
42
```

 Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

En Ruby, podemos simplificar la manera de escribir un `if` dentro un `else` con `elsif`. Por ejemplo este código:

```
def self.nota_conceptual(nota)
  if nota > 8
    "Sobresaliente"
  else
    if nota > 6
      "Satisfactoria"
    else
      "No satisfactoria"
    end
  end
end
```



Lo podemos escribir:

```
def self.nota_conceptual(nota)
  if nota > 8
    "Sobresaliente"
  elsif nota > 6
    "Satisfactoria"
  else
    "No satisfactoria"
  end
end
```



Antes de seguir, ¿te animás a editar tu solución para que use `elsif`?

Esta guía fue desarrollada por Federico Aloí y [muchas personas más](#), 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](#)





Llegó Pepo

Pepo es un gorrión que también sabe comer, volar y hacer lo que quiera, pero lo hace de manera diferente a **Pepita**.

5. Llegó Pepo

- **comer alpiste:** el aparato digestivo de **Pepo** no anda muy bien, por eso solo puede aprovechar la mitad del alpiste que come. Por ejemplo, si come 20 gramos de alpiste, su energía solo aumenta en 10.
- **volar en círculos:** gasta 15 unidades de energía si está pesado y 5 si no lo está. Decimos que está pesado si su energía es mayor a 1100.
- **hacer lo que quiera:** como siempre tiene hambre, aprovecha y come 120 gramos de alpiste.

Ah, y al igual que **Pepita**, su energía comienza en **1000**.

Definí a **Pepo** según las reglas anteriores. Te dejamos el código de **Pepita** para usar como base, modificá y borra las partes que no correspondan.

💡 ¡Dame una pista!

 Solución  Biblioteca  Consola

```
1 module Pepo
2   @energia = 1000
3
4   def self.energia
5     @energia
6   end
7
8   def self.volar_en_circulos!
9     if self.pesado?
10      @energia -= 15
11    else
12      @energia -= 5
13    end
```



```
14 end
15
16 def self.comer_alpiste!(gramos)
17   @energia += gramos / 2
18 end
19
20 def self.pesado?
21   self.energia > 1100
22 end
23
24 def self.hacer_lo_que_quiera!
25   self.comer_alpiste!(120)
26 end
27
28 end
29
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Genial, ya tenemos dos aves con las cuales trabajar y que además **comparten una interfaz**: ambas entienden los mensajes `comer_alpiste!(gramos)`, `volar_en_circulos!` y `hacer_lo_que_quiera!`.

Veamos qué podemos hacer con ellas...

Esta guía fue desarrollada por Federico Aloí y [muchas personas más](#), 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](#)





¡A entrenar!

Nuestras aves quieren presentarse a las próximas Olimpiadas, y para eso necesitan ~~ejercitar~~ ^{6. ¡A entrenar!} un poco.

Para ayudarnos en esta tarea conseguimos a **Pachorra**, un ex entrenador de fútbol que ahora se dedica a trabajar con aves. Él diseñó una rutina especial que consiste en lo siguiente:

- Volar en círculos 10 veces.
- Comer un puñado de 30 gramos de alpiste.
- Volar en círculos 5 veces.
- Como premio, que el ave haga lo que quiera.

Creá a **Pachorra**, el entrenador de aves, y hacé que cuando reciba el mensaje **entrenar_ave!** haga que **Pepita** realice su rutina (si, solo puede entrar a Pepita, pero lo solucionaremos pronto).

Para que no moleste, movimos el código de **Pepita** a la **Biblioteca**.

💡 ¡Dame una pista!

 Solución  Biblioteca  _ Consola

```
1 module Pachorra
2
3   def self.entrenar_ave!
4     10.times {Pepita.volar_en_circulos! }
5     Pepita.comer_alpiste!(30)
6     5.times {Pepita.volar_en_circulos!}
7     Pepita.hacer_lo_que_quiera!
8   end
9 end
10
```

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Aunque lo que hiciste funciona, es bastante rígido: para que `Pachorra` pueda entrenar a otro pájaro hay que modificar el **método** `entrenar_ave!` y cambiar el objeto al que le envía los mensajes.

¡Mejoremos eso entonces!

Esta guía fue desarrollada por Federico Aloí y [muchas personas más](#), 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](#)





Pachorra todoterreno

•

Como imaginabas, **Pachorra** puede entrenar cualquier tipo de aves, aunque para que no haya problemas, solo entrena de a una a la vez.

Antes de empezar a entrenar, debe firmar un contrato con el ave. Esto, por ejemplo, lo haríamos de la siguiente manera:

```
Pachorra.firmar_contrato! Pepita # ahora el ave de Pachorra es Pepita
```

Cada vez que firmamos un contrato cambiamos el **ave** que entrenará **Pachorra**, por lo cual es necesario recordar cuál es ya que a ella le enviaremos mensajes:

```
Pachorra.entrenar_ave! # acá entrena a Pepita
Pachorra.firmar_contrato! Pepo # ahora el ave de Pachorra es Pepo
Pachorra.entrenar_ave! # ahora entrena a Pepo
```

Agregale a **Pachorra** el método **firmar_contrato!(ave)**, de forma tal que cuando le enviemos el mensaje **entrenar_ave!** haga entrenar al último ave con el que haya firmado contrato.

💡 ¡Dame una pista!

 Solución  Consola

```
1 module Pachorra
2
3   def self.firmar_contrato!(ave)
4     @ave_contratada = ave
5   end
6
7   def self.entrenar_ave!
8     10.times {@ave_contratada.volar_en_circulos!}
9     @ave_contratada.comer_alpiste!(30)
10    5.times {@ave_contratada.volar_en_circulos!}
11    @ave_contratada.hacer_lo_que_quiera!
```

```
12   end
13   end
14
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Una forma posible de cambiar el objeto al que le enviamos mensajes es **modificando el valor de un atributo**, como estamos haciendo en este ejemplo.

Esta guía fue desarrollada por Federico Aloí y [muchas personas más](#), 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](#)





Una golondrina diferente

•

¿Te acordás de **Norita**, la amiga de **Pepita**? Resulta que ella también quiere empezar a entrenar, y su código es el siguiente:

```
module Norita
  @energia = 500

  def self.volar_en_circulos!
    @energia -= 30
  end

  def self.comer_alpiste!(gramos)
    @energia -= gramos
  end
end
```

Pero, ¿podrá entrenar con **Pachorra**?

Probalo en la consola, enviando los siguientes mensajes:

```
> Pachorra.firmar_contrato! Norita
> Pachorra.entrenar_ave!
```

>



Esta guía fue desarrollada por Federico Aloï y [muchas personas más](#), 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](#)





Un entrenamiento más duro

Analicemos el error:

9. Un entrenamiento más duro

```
> Pachorra.entrenar_ave!  
undefined method `hacer_lo_que_quiera!' for Norita:Module (NoMethodError)
```

En criollo, lo que dice ahí es que `Norita` no entiende el mensaje `hacer_lo_que_quiera!`, y por eso `Pachorra` no la puede entrenar; este mensaje forma parte de su rutina.

Miremos ahora el método `entrenar_ave!` de `Emilce`, una entrenadora un poco más estricta:

```
module Emilce  
  def self.entrenar_ave!  
    53.times { @ave.volar_en_circulos! }  
    @ave.comer_alpiste! 8  
  end  
end
```

¿Podrá `Norita` entrenar con `Emilce`? ¿Y `Pepita`? ¿Y `Pepo`?

Probalo en la consola y completá el código con `true` (verdadero) o `false` (falso) según corresponda para cada ave.

💡 ¡Dame una pista!

Solución Consola

```
1 norita_puede_entrenar_con_pachorra = false  
2 norita_puede_entrenar_con_emilce = true  
3  
4 pepita_puede_entrenar_con_pachorra = true  
5 pepita_puede_entrenar_con_emilce = true  
6  
7 pepo_puede_entrenar_con_pachorra = true  
8 pepo_puede_entrenar_con_emilce = true
```

Enviar

✅ ¡Muy bien! Tu solución pasó todas las pruebas

Según las rutinas que definen, cada entrenador/a solo puede trabajar con ciertas aves:

- `Pachorra` puede entrenar a cualquier ave que entienda `volar_en_circulos!`, `comer_alpiste!(gramos)` y `hacer_lo_que_quiera!`.
- `Emilce` puede entrenar a cualquier ave que entienda `volar_en_circulos!` y `comer_alpiste!(gramos)`.

Dicho de otra manera, la rutina nos define cuál debe ser la **interfaz** que debe respetar un objeto para poder ser utilizado.

Esta guía fue desarrollada por Federico Aloí y [muchas personas más](#), 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](#)





¿¿Polimor-qué??



¿Qué pasa si dos objetos, como [Pepita](#), [Norita](#) o [Pepo](#) son capaces de responder a un mismo mensaje? Podemos intercambiar un objeto por otro sin notar la diferencia, como experimentaste recién.

Este concepto es fundamental en objetos, y lo conocemos como **polimorfismo**. Decimos entonces que dos objetos son **polimórficos** cuando pueden responder a un mismo conjunto de mensajes y hay un tercer objeto que los usa indistintamente. Dicho de otra forma, dos objetos son **polimórficos para un tercer objeto** cuando este puede enviarles los mismos **mensajes**, sin importar cómo respondan o qué otros mensajes entiendan.

En nuestro caso:

- [Pepita](#), [Norita](#) y [Pepo](#) son polimórficas para [Emilce](#).
- [Pepita](#), [Norita](#) y [Pepo](#) **no** son polimórficas para [Pachorra](#).
- [Pepita](#) y [Pepo](#) son polimórficas para [Pachorra](#).

Esta guía fue desarrollada por Federico Aloí y [muchas personas más](#), 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](#)





Forzando el polimorfismo

Bueno, ya entendimos que para el caso de `Pachorra`, `Norita` no es **polimórfica** con las otras aves, pero... ¿podremos hacer algo al respecto?

11. Forzan

¡Claro que sí! Podemos agregarle los mensajes que le faltan, en este caso `hacer_lo_que_quiera!`.

¿Y qué hace `Norita` cuando le decimos que haga lo que quiera? Nada.

Modificá a `Norita` para que pueda entrenar con `Pachorra`.

Solución Biblioteca Consola

```
1 module Norita
2   @energia = 500
3
4   def self.energia
5     @energia
6   end
7
8   def self.volar_en_circulos!
9     @energia -= 30
10  end
11
12  def self.comer_alpiste!(gramos)
13    @energia -= gramos
14  end
15
16  def self.hacer_lo_que_quiera!
17    end
18
19 end
20
```

Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Aunque parezca que no tiene mucho sentido, es común que trabajando con objetos necesitemos forzar el **polimorfismo** y hagamos cosas como estas.

En este caso le agregamos a `Norita` un mensaje que no hace nada, con el único objetivo de que sea polimórfica con sus compañeras aves.

Esta guía fue desarrollada por Federico Aloí y [muchas personas más](#), 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](#)





Empieza el set

En los ejercicios anteriores, le habíamos incluido a **Pachorra** y **Emilce** un mensaje `firmar_contrato!(ave)` que modificaba su **estado**, es decir, alguno de sus **atributos**. A estos mensajes que solo modifican un atributo los conocemos con el nombre de **setters**, porque vienen del inglés **set** que significa establecer, ajustar, fijar.

Para estos casos, solemos utilizar una convención que se asemeja a la forma que se modifican los atributos desde el propio objeto, pudiendo ejecutar el siguiente código desde una consola:

```
Emilce.ave = Pepita
```

Esto se logra definiendo el método `ave=`, todo junto, como se ve a continuación:

```
module Emilce
  def self.ave=(ave_nueva)
    @ave = ave_nueva
  end

  def self.entrenar_ave!
    53.times { @ave.volar_en_circuitos! }
    @ave.comer_alpiste!(8)
  end
end
```

¿Te animás a cambiar el código de **Pachorra** para que siga esta convención?

Solución > Consola

```
1 module Pachorra
2   def self.ave=(ave_nueva)
3     @ave = ave_nueva
4   end
5
6   def self.entrenar_ave!
7     10.times { @ave.volar_en_circuitos! }
8     @ave.comer_alpiste! 30
9     5.times { @ave.volar_en_circuitos! }
10    @ave.hacer_lo_que_quiera!
11  end
12 end
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Como ya te habíamos contado en una lección anterior, a estos métodos que solo sirven para acceder o modificar un atributo los llamamos **métodos de acceso** o **accessors**. Repasando, los **setters** son aquellos métodos que *establecen* el valor del atributo. Mientras que los **getters** son aquellos que *devuelven* el valor del atributo.

La convención en Ruby para estos métodos es:

- Los **setters** deben llevar el mismo nombre del atributo al que están asociados, agregando un `=` al final.
- Los **getters** usan exactamente el mismo nombre que el atributo del cual devuelven el valor pero sin el `@`.
- Aquellos **getters** que devuelven el valor de un atributo booleano llevan `?` al final.

Esta guía fue desarrollada por Federico Aloí y [muchas personas más](#), 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](#)





El encapsulamiento

Ya aprendiste cómo crear **getters** y **setters** para un atributo, pero ¿siempre vamos a querer ambos?

La respuesta es que no, y a medida que desarrolles más programas y dominios diferentes tendrás que construir tu propio criterio para decidir cuándo sí y cuándo no.

Por ejemplo, ¿qué pasaría si a **Pepita** le agregáramos un setter para la ciudad? Podríamos cambiarla en cualquier momento de nuestro programa ¡y no perdería energía! Eso va claramente en contra de las reglas de nuestro dominio, y no queremos que nuestro programa lo permita.

Te dejamos en la **Biblioteca** el código que modela a **Manuelita**, una tortuga viajera. Algunos de sus atributos pueden ser leídos, otros modificados y otros ambas cosas.

Completá las listas de `atributos_con_getter` y `atributos_con_setter` mirando en la definición de Manuelita qué tiene programado como setter y que como getter.

💡 ¡Dame una pista!

Solución

Biblioteca

Consola

```
1 atributos = %w(  
2   energia  
3   ciudad  
4   mineral_preferido  
5   donde_va  
6 )  
7  
8 atributos_con_getter = %w(  
9   ciudad  
10  energia  
11  mineral_preferido  
12 )  
13  
14 atributos_con_setter = %w(  
15  mineral_preferido  
16  donde_va  
17 )
```

Enviar

✅ ¡Muy bien! Tu solución pasó todas las pruebas

Si hacemos bien las cosas, quien use nuestros objetos sólo verá lo que necesite para poder interactuar con ellos. A esta idea la conocemos como **encapsulamiento**, y es esencial para la separación de **responsabilidades** de la que veníamos hablando.

Será tarea tuya (y de tu equipo de trabajo, claro) decidir qué atributos exponer en cada objeto.

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





Vamos terminando

Vamos a empezar a repasar todo lo que aprendiste en esta lección, te vamos a pedir que modelés a nuestro amigo **Inodoro**, un gaucho solitario de la pampa argentina. Fiel al estereotipo, **Inodoro** se la pasa tomando **mate**, y siempre lo hace con algún compinche; ya sea **Eulogia**, su compañera o **Mendieta**, su perro parlante.

Tu tarea será completar el código que te ofrecemos, definiendo los métodos incompletos y agregando los getters y setters necesarios para que sea posible:

- Consultar cuánta cafeína en sangre tiene **Inodoro**.
- Consultar al **compinche** de **Inodoro**.
- Modificar al **compinche** de **Inodoro**.
- Consultar si **Eulogia** está enojada.
- Consultar cuántas ganas de hablar tiene **Mendieta**.
- Modificar las ganas de hablar de **Mendieta**.

💡 ¡Dame una pista!

🔍 Solución

🔍 Consola

```
1 module Inodoro
2   @cafeina_en_sangre = 90
3
4   #(esto es un getter)
5   def self.cafeina_en_sangre
6     @cafeina_en_sangre
7   end
8
9   #(esto es un getter)
10  def self.compinche
11    @compinche
12  end
13
14  #(esto es un setter)
15  def self.compinche=(nuevo_compinche)
16    @compinche = nuevo_compinche
17  end
18
19 end
20
21
22 module Eulogia
23   @enojada = false
24
25   def self.enojada?
26     @enojada
27   end
28
29 end
30
31
32 module Mendieta
33   @ganas_de_hablar = 5
34
35   def self.ganas_de_hablar
36     @ganas_de_hablar
37   end
38
39   def self.ganas_de_hablar=(ganas)
```



```
40 @ganas_de_hablar = ganas
41 end
42
43 end
44
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

¡Excelente! Parece que los getters y setters quedaron claros.

Para finalizar esta lección vamos a repasar lo que aprendimos de polimorfismo.

Esta guía fue desarrollada por Federico Aloí y [muchas personas más](#), 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](#)





¡Se va la que falta!

Para finalizar el repaso vamos a modelar el comportamiento necesario para que `Inodoro` pueda tomar mate con cualquiera de sus compinches... ¡Polimórficamente!

- Cuando `Inodoro` toma mate aumenta en 10 su cafeína en sangre y su compinche recibe un mate.
- Al recibir un mate, `Eulogia` se enoja porque `Inodoro` siempre le da mates fríos.
- Por su parte, `Mendieta` se descompone cuando recibe un mate, porque bueno... es un perro. Esto provoca que no tenga nada de ganas de hablar (o en otras palabras, que sus `ganas_de_hablar` se vuelvan `0`).

Definí los métodos `tomar_mate!`, en `Inodoro`, y `recibir_mate!` en `Eulogia` y `Mendieta`.

💡 ¡Dame una pista!

🔍 Solución

>_ Consola

```
1 module Inodoro
2   @cafeina_en_sangre = 90
3
4   #(esto es un getter)
5   def self.cafeina_en_sangre
6     @cafeina_en_sangre
7   end
8
9   #(esto es un setter)
10  def self.compinche=(nuevo_compinche)
11    @compinche = nuevo_compinche
12  end
13
14  #(esto es un getter)
15  def self.compinche
16    @compinche
17  end
18
19  #(esto es un setter)
20  def self.tomar_mate!
21    @cafeina_en_sangre += 10
22    @compinche.recibir_mate!
23  end
24
25 end
26
27
28
29
30 module Eulogia
31   @enojada = false
32
33   def self.enojada?
34     @enojada
35   end
36
37   #(esto es un setter)
38   def self.recibir_mate!
39     @enojada = true
40   end
41
42 end
43
44
```

```
45 module Mendieta
46   @ganas_de_hablar = 5
47
48   def self.ganas_de_hablar
49     @ganas_de_hablar
50   end
51
52   def self.ganas_de_hablar=(ganas)
53     @ganas_de_hablar = ganas
54   end
55
56   #(esto es un setter)
57   def self.recibir_mate!
58     @ganas_de_hablar = 0
59   end
60
61 end
62
63
64
```

► Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Esta guía fue desarrollada por Federico Aloí y [muchas personas más](#), 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](#)

