



500793827 Argentina Programa 3 Ruby comprimido

Programación I (Universidad Abierta Interamericana)



Capítulo 3: Programación con Objetos

El paradigma de objetos, a veces también conocido como orientado a objetos nos propone solucionar problemas y modelar nuestra realidad empleando objetos que se comunican entre ellos intercambiando mensajes. ¡Adentrémonos en el mundo de los objetos y Ruby!

Lecciones

1. Objetos y mensajes

- 1. Fijando nuestro objetivo
- 2. ¡Hola Pepita!
- 3. Un mundo de objetos
- 4. El derecho a la Identidad
- 5. Mensajes, primera parte
- 6. Mensajes, segunda parte
- 7. No entendí...
- 8. Un poco de sintaxis
- 9. Interfaz
- 10. Hacer versus Devolver
- 11. Tu primer programa con objetos
- 12. ¿Quién te entiende?
- 13. Interfaces compartidas
- 14. Argumentos
- 15. Más argumentos
- 16. Mensajes por todas partes
- 17. Recapitulando

- 12. Cambiando la referencia
- 13. El encapsulamiento
- 14. Vamos terminando
- 15. ¡Se va la que falta!

4. Colecciones

- 1. Entrando en Calor
- 2. Creando una lista
- 3. Algunos mensajes básicos
- 4. Mejorando la Biblioteca
- 5. ¿Bloques? ¿Eso se come?
- 6. Bloques con parámetros
- 7. Filtrando quienes cumplen
- 8. El que busca encuentra
- 9. ¿Alguno cumple? ¿Todos cumplen?
- 10. El viejo y querido map
- 11. ¿Cuántos cumplen? ¿Cuánto suman?
- 12. Jugando a todo

2. Definiendo objetos: métodos y estado

- 1. Creando a Pepita
- 2. Pepita, ¿me entendés?
- 3. Los mejores, los únicos, los métodos en objetos
- 4. Perdiendo energía
- 5. Atributos
- 6. Asignaciones y referencias
- 7. Conociendo el país
- 8. Leyendo el estado
- 9. Cuestión de estado
- 10. ¿Dónde estás?
- 11. Volando alto
- 12. Delegar es bueno
- 13. ¿Es mi responsabilidad?

5. Referencias

- 1. Variables
- 2. Las variables son referencias
- 3. Referencias implícitas
- 4. Múltiples referencias
- 5. Identidad, revisada
- 6. Equivalencia
- 7. Objetos bien conocidos
- 8. Atributos y parámetros
- 9. Objetos compartidos
- 10. Para cerrar

6. Clases e Instancias

- 1. Zombi caminante
- 2. Atacando un zombi
- 3. Otro zombi caminante
- 4. ¡¿Vivos?!
- 5. Clases
- 6. Instancias
- 7. Al menos tenemos salud
- 8. Inicializando instancias
- 9. Ahora sí: invasión
- 10. Al menos tenemos (menos) salud
- 11. Súper zombi
- 12. ...

3. Polimorfismo y encapsulamiento

- 1. ¿Pepita está feliz?
- 2. Reencuentro alternativo
- 3. Repitamos qué pasa si no
- 4. Voy a hacer, pero como yo quiero
- 5. Llegó Pepo
- 6. ¡A entrenar!
- 7. Pachorra todoterreno
- 8. Una golondrina diferente
- 9. Un entrenamiento más duro
- 10. ¿Polimor-qué??
- 11. Forzando el polimorfis



7. Herencia

1. Auto
2. Moto
3. Clase abstracta
4. Medio de transporte
5. Todo pesa
6. Peso repetido
7. Bicicleta
8. Muy pesada
9. Redefiniendo peso
10. El regreso de los zombis
11. Herencia zombie
12. Micro
13. Suben y bajan
14. La pesada herencia
15. Súper ataque
16. Camiones muy especiales

8. Excepciones

1. ¡Sin energía!
2. Sólo Volar Si...
3. Una falla silenciosa
4. ¡Fallar!
5. Lanzando excepciones
6. Abortando la evaluación
7. Paren todo
8. El orden importa
9. Estudiando a pepita
10. Un buen mensaje



Fijando nuestro objetivo

Anteriormente hablamos de los *paradigmas de programación*. En este capítulo vamos a ver otra forma de pensar el mundo de la programación.

El paradigma de programación con objetos o programación *orientada a objetos* nos propone tratar con...¡Adiviná! Sí, nos permite trabajar con objetos. 😊

En este video te contamos qué son esos famosos objetos y cómo interactúan entre sí.

Introducción a Programación con Objetos con Ruby





¡Hola Pepita!

Para empezar en este mundo, conozcamos a **Pepita**, una [golondrina](http://es.wikipedia.org/wiki/Hirundo_rustica) (http://es.wikipedia.org/wiki/Hirundo_rustica).



Pepita, además de ser un ave que come y vuela (como todo pájaro), es un objeto, que vive en el *mundo de los objetos*, al cual conocemos como **ambiente**.

¿No nos creés que **Pepita** está viva y es un objeto? Escribí en la consola **Pepita** y fijate qué sucede. Cuando te convenzas, pasá al siguiente ejercicio.

☒ [;Dame una pista!](#)

La **consola** es el recuadro que tenés a la derecha del texto, y sirve para hacer pruebas rápidas. ✓

Usarla es muy sencillo: escribí lo que querés probar y luego presioná la tecla **Enter** para ver su resultado. Si alguna vez usaste alguna terminal, como la de [GNU/Linux](https://es.wikipedia.org/wiki/GNU/Linux) (<https://es.wikipedia.org/wiki/GNU/Linux>), la mecánica es la misma.

☒ **Pepita**

=> **Pepita**

☒

☒



Un mundo de objetos

Como vimos, Pepita es un objeto. Pero Pepita no está sola en este mundo. ¡Hay muchos más!

Por ejemplo, existe otra golondrina, llamada Norita, que también vive en este ambiente.

Como ya te vendrás dando cuenta, en este **paradigma** bajo el cual estamos trabajando absolutamente todo es un objeto: también los números, las cadenas de texto (o strings) y los booleanos.

¡Probalo! Hacé las siguientes consultas en la consola:

```
☒ Pepita
☒ Norita
☒ 87
☒ 'hola mundo'
☒ true
```

De todas formas tené cuidado. A Pepita y Norita las creamos por vos. Y los números y booleanos vienen de regalo. Si probás con otra cosa, como por ejemplo Felix, te va a tirar un error.

```
☒ Norita
=> Norita
☒ 87
=> 87
☒ 'hola mundo'
=> "hola mundo"
☒ true
=> true
```



El derecho a la Identidad

Un aspecto muy importante de los objetos es que tienen **identidad**: cada objeto sabe quién es y gracias a esto sabe también que es diferente de los demás. Por ejemplo, Pepita sabe que ella es diferente de Norita, y viceversa.

En Ruby, podemos comparar por identidad a dos objetos utilizando el operador `==` de la siguiente forma:

```
☒ Pepita == Norita
```

¡Intentalo por tu propia cuenta! Ejecutá las siguientes pruebas en la consola:

```
☒ Pepita == Norita
☒ Norita == Pepita
☒ Norita == Norita
☒ "hola" == "chau"
```

☒ ¡Dame una pista!

¿Te cansaste de escribir? Te dejamos un truquito para usar la consola más rápido: si apretás la flecha para arriba ↑, se repite lo último que escribiste. ☺

```
☒ Pepita == Norita
=> false
☒ Norita == Pepita
=> false
☒ Norita == Norita
=> true
☒ "hola" == "chau"
=> false
```



Mensajes, primera parte

Ya entendimos que en un ambiente hay objetos, y que cada uno de ellos tiene *identidad*: sabe que es diferente de otro.

Pero esto no parece ser muy útil. ¿Qué cosas sabrá hacer una golondrina como Pepita? ¿Sabrá, por ejemplo, cantar! ? 🎵

Averigualo: envíale un mensaje `cantar!` y fijate qué pasa...

☒ `Pepita.cantar!`

☒ ¡Dame una pista!

El signo de exclamación ! es parte del nombre del mensaje, no te olvides de ponerlo. 😊

```
☒ Pepita.cantar!  
=> "pri pri pri"  
☒
```



Mensajes, segunda parte

Ehhh, ¿qué acaba de pasar acá? 😊

Para comunicarnos con los objetos, debemos enviarles **mensajes**. Cuando un objeto recibe un mensaje, este responde *haciendo algo*. En este caso, Pepita produjo el sonido de una golondrina: pri pri pri ...imagine acá que escuchamos *este sonido* (<https://www.youtube.com/watch?v=NteRoisnwAI>)... 😊.

¿Qué mas sabrá hacer Pepita ? ¿Sabrá, por ejemplo, bailar! ? 🎶

¡Descubrámoslo! Enviale el mensaje `bailar!`

☒ `Pepita.bailar!`



☒ `Pepita.bailar!`

`undefined method `bailar!' for Pepita:Module (NoMethodError)`

☒





No entendí...

¡Buu, Pepita no sabía bailar! 😊

En el mundo de los objetos, sólo tiene sentido enviarle un mensaje a un objeto si lo entiende, es decir, si sabe hacer algo como reacción a ese mensaje. De lo contrario, se lanzará un error un poco feo (y en inglés 😱) como el siguiente:

```
undefined method `bailar!' for Pepita:Module
```

Descubramos qué otras cosas sabe hacer Pepita. Probá enviarle los siguientes mensajes y fijate cuáles entiende y cuáles no ¡y anotalo! 📝 Este conocimiento nos servirá en breve.

- ☒ Pepita.pasear!
- ☒ Pepita.energia
- ☒ Pepita.comer_lombriz!
- ☒ Pepita.volar_en_circulos!
- ☒ Pepita.se_la_banca?

```
☒ Pepita.pasear!  
undefined method `pasear!' for Pepita:Module (NoMethodError)  
☒ Pepita.energia  
=> 100  
☒ Pepita.comer_lombriz!  
=> nil  
☒ Pepita.volar_en_circulos!  
=> nil
```



Un poco de sintaxis

¡Pausa! Analicemos la sintaxis del envío de mensajes:

1. Pepita.energia es un envío de mensaje, también llamado **colaboración**;
2. energia es el **mensaje**;
3. energia es el nombre del mensaje (en este caso es igual, pero ya veremos otros en los que no);
4. Pepita es el objeto receptor del mensaje.

⚠ Es importante respetar la sintaxis del envío de mensajes. Por ejemplo, las siguientes NO son colaboraciones válidas, porque no funcionan o no hacen lo que deben:

```
✗ energia
✗ Pepita energia
✗ Pepita..energia
```

¿Eh, no nos creés? 😊 ;Probalas!

```
✗ energia
undefined local variable or method `energia' for main:Object (NameError)
✗ Pepita energia
undefined local variable or method `energia' for main:Object (NameError)
✗ Pepita..energia
undefined local variable or method `energia' for main:Object (NameError)
✗
```

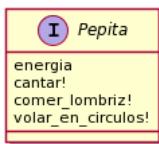


Interfaz

Como vimos, un objeto puede entender múltiples mensajes; a este conjunto de mensajes que podemos enviarle lo denominamos **interfaz**. Por ejemplo, la interfaz de Pepita es:

- `energia` : nos dice cuanta energía tiene (un número);
- `cantar!` : hace que cante;
- `comer_lombriz!` : hace que coma una lombriz;
- `volar_en_circulos!` : hace que vuele en circulos.

Lo cual también se puede graficar de la siguiente forma:



¡Un momento! ¿Por qué algunos mensajes terminan en `!` y otros no? Enviá nuevamente esos mensajes. Fijate qué devuelve cada uno (lo que está a la derecha del `=>`) y tratá de descubrir el patrón.

☒ ¡Dame una pista!

`nil` es la forma que tenemos en Ruby de representar a "la nada" (que, casualmente, ¡también es un objeto!).

```
☒ Pepita.energia  
=> 100  
☒ Pepita.cantar!  
=> "pri pri pri"  
☒ Pepita.comer_lombriz!  
=> nil  
☒ Pepita.volar_en_circulos!  
=> nil
```



Hacer versus Devolver

Cuando se envía un mensaje a un objeto, y este lo entiende, puede reaccionar de dos formas diferentes:

- Podría *producir un efecto*, es decir hacer algo. Por ejemplo, el mensaje `cantar!` reproduce el sonido del canto de `Pepita`.
- O también podría *devolver otro objeto*. Por ejemplo el mensaje `energia` devuelve siempre un número.

En realidad, un mensaje podría reaccionar con una combinación de las formas anteriores: tener un efecto y devolver algo. Pero esto es una **muy mala idea**.

¿Y qué hay de los mensajes como `comer_lombriz!` y `volar_en_circulos!`? ¿Hicieron algo? ¿Qué clase de efecto produjeron? ¿Devuelve `energia` siempre lo mismo? 😐🔗

Descubrilo: envíale a `Pepita` esos tres mensajes varias veces en distinto orden y fijate si cambia algo.

```
☒ Pepita.volar_en_circulos!
=> nil
☒ Pepita.energia
=> 90
☒ Pepita.comer_lombriz!
=> nil
☒ Pepita.energia
=> 110
```



Tu primer programa con objetos

¡Exacto! El efecto que producen los mensajes `comer_lombriz!` y `volar_en_circulos!` es el de alterar la energía de `Pepita`. En concreto:

- `comer_lombriz!` hace que la energía de `Pepita` aumente en 20 unidades;
- `volar_en_circulos!` hace que la energía de `Pepita` disminuya en 10 unidades.

Como convención, a los mensajes **con efecto** (es decir, que *hacen algo*) les pondremos un signo de exclamación `!` al final.

Veamos si se entiende: escribí un primer programa que consista en hacer que `Pepita` coma y vuela hasta quedarse con 150 unidades de energía. Acordate que `Pepita` arranca con la energía en 100.

¡Dame una pista!

No te olvides de que *siempre* tenés que poner el objeto receptor del mensaje. Por ejemplo esto es válido:

`Pepita.energia`

pero esto no:

`energia`

Solución Consola

```
1 # Escribí acá los mensajes que quieras mandarle a Pepita, uno debajo del otro.  
2 Pepita.energia  
3 Pepita.comer_lombriz!  
4 Pepita.comer_lombriz!  
5 Pepita.volar_en_circulos!  
6 Pepita.comer_lombriz!  
7 Pepita.energia
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



Podemos sacar dos conclusiones:

1. Los objetos no reaccionan necesariamente siempre igual a los mismos mensajes. Podrían hacer cosas diferentes, o en este caso, devolver objetos distintos.
2. ¡Un programa es simplemente una secuencia de envío de mensajes!

Esta guía fue desarrollada por Federico Aloí, Franco Bulgarelli, Ariel Umansky bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





¿Quién te entiende?

Ya vimos que un objeto puede entender múltiples mensajes, y esos mensajes conforman su interfaz.

¿Pero podría haber más de un objeto que entienda los mismos mensajes?

A Pepita ya la conocemos bien: canta, come, etc. Su amiga Norita, por otro lado, no aprendió nunca a decirnos su energía. Y Mercedes es una reconocida cantora.

Usando la consola, averiguá cuál es la interfaz de cada una de ellas, y completá el listado de mensajes que cada una entiende en el editor.

☒ ¡Dame una pista!

Buscá la **Consola** en la pestaña de al lado de tu **Solución** y probá de enviarle a cada objeto los mismos mensajes que Pepita entiende.

Cuando descubras cuáles son, completá la solución con el mismo formato que viene la de Pepita.

☒ Solución ☒ Consola

```
1 interfaz_pepita = %w(
2   energia
3   cantar!
4   comer_lombriz!
5   volar_en_circulos!
6 )
7
8 interfaz_norita = %w(
9   cantar!
10  comer_lombriz!
11  volar_en_circulos!
12 )
13
14 interfaz_mercedes = %w(
15   cantar!
16 )
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas

¡Así es! Puede haber más de un objeto que entienda el mismo mensaje. Notá que sin embargo no todos los objetos están obligados a reaccionar de igual forma ante el mismo mensaje:

```
Pepita.cantar!
=> "pri pri pri"

Norita.cantar!
=> "priiiip priiiip"

Mercedes.cantar!
=> "♪ una voz antigua de viento y de sal ♪"
```

Esto significa que dos o más objetos pueden entender un mismo mensaje, pero pueden **comportarse** de formas diferentes. Ya hablaremos más de esto en próximas lecciones.



Interfaces compartidas

Veamos si queda claro, siendo que las interfaces de `Norita`, `Pepita` y `Mercedes` son las siguientes:



Esto significa que comparten algunos mensajes y otros no. ¿Qué interfaces comparten entre ellas?

Completá el código en el editor.

¡Dame una pista!

Recordá que la interfaz es el conjunto de mensajes que un objeto entiende. Por lo tanto, si queremos ver cual interfaz comparten dos objetos, tenemos que pensar en la intersección entre los conjuntos de mensajes de cada uno (es decir, aquellos que son iguales).

Solución Consola

```

1 # ¿Qué interfaz comparten Mercedes y Norita?
2 interfaz_compartida_entre_mercedes_y_norita = %w(
3     cantar!
4 )
5
6 # ¿Qué interfaz comparten Pepita y Norita?
7 interfaz_compartida_entre_pepita_y_norita = %w(
8     cantar!
9     comer_lombriz!
10    volar_en_circulos!
11 )
12
13 # ¿Qué interfaz comparten Mercedes, Norita y Pepita?
14 interfaz_compartida_entre_todas = %w(
15     cantar!
16 )

```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



Argumentos

Para hacer las cosas más interesantes, vamos a necesitar mensajes más complejos. 😊

Por ejemplo, si queremos que `Pepita` coma una cierta cantidad de alpiste que no sea siempre la misma, necesitamos de alguna manera indicar cuál es esa cantidad. Esto podemos escribirlo de la siguiente forma:

```
Pepita.comer_alpiste!(40)
```

Allí, `40` es un *argumento* del mensaje, representa en este caso que vamos a alimentar a pepita con 40 gramos de alpiste. Un mensaje podría tomar más de un argumento, separados por coma.

Probá enviar los siguientes mensajes:

```
☒ Pepita.volar_hacia!(Iruya)
☒ Pepita.comer_alpiste!(39)
☒ Pepita.comer_alpiste!(6, Norita)
```

```
☒ Pepita.volar_hacia!(Iruya) 
=> nil
☒ Pepita.comer_alpiste!(39)
=> nil
☒ Pepita.comer_alpiste!(6, Norita)
wrong number of arguments (given 2, expected 1) (ArgumentError)
☒
```



Más argumentos

Como ves, si enviás un mensaje con una cantidad incorrecta de argumentos...

```
☒ Pepita.comer_alpiste!(6, Norita)
# wrong number of arguments (2 for 1) (ArgumentError)
```

...el envío del mensaje también fallará.

Dicho de otra forma, **un mensaje queda identificado no sólo por su nombre sino también por la cantidad de parámetros que tiene**: no es lo mismo `comer_alpiste!` que `comer_alpiste!(67)` que `comer_alpiste!(5, 6)`, son todos mensajes distintos. Y en este caso, `Pepita` sólo entiende el segundo.

Veamos si va quedando claro: escribí un programa que haga que `Pepita` coma 500 gramos de alpiste, vuele a `Iruya` (<https://es.wikipedia.org/wiki/Iruya>), y finalmente vuelva a `Obera` (<https://es.wikipedia.org/wiki/Ober%C3%A1>).

☒ ¡Dame una pista!

Tené en cuenta que nuestra golondrina entiende también los siguientes mensajes:

- `volar_hacia!`, que espera como argumento una ciudad;
- `comer_alpiste!`, que espera como argumento una cantidad de gramos de alpiste;

Y que además, existen los objetos `Iruya` y `Obera`.

☒ Solución ☒ Consola

```
1 Pepita.comer_alpiste!(500)
2 Pepita.volar_hacia!(Iruya)
3 Pepita.volar_hacia!(Obera)
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas

¡Perfecto! 🎉

Un detalle: en Ruby, *a veces*, los paréntesis sonopcionales. Para no confundirte, vamos a omitirlos **sólo cuando el mensaje no tenga argumentos** (como en `Pepita.energia`) y los pondremos siempre que los tenga (como en `Pepita.volar_hacia!(Obera)`).



Mensajes por todas partes

Es fácil ver que en `Pepita.volar_hacia!(Barreal)` el objeto receptor es `Pepita`, el mensaje `volar_hacia!` y el argumento `Barreal` (<https://es.wikipedia.org/wiki/Barreal>); pero ¿dónde queda eso de objeto y mensaje cuando hacemos, por ejemplo, `2 + 3`?

Como ya dijimos, todas nuestras interacciones en un ambiente de objetos ocurren enviando mensajes y las operaciones aritméticas **no son la excepción** a esta regla.

En el caso de `2 + 3` podemos hacer el mismo análisis:

- el objeto receptor es `2`;
- el mensaje es `+`;
- el argumento es `3`.

Y de hecho, ¡también podemos escribirlo como un envío de mensajes convencional!

Probá en la consola los siguientes envíos de mensajes:

```
☒ 5.+(6)
☒ 3.<(27)
☒ Pepita.==(Norita)
```

```
☒ 5.+(6)
=> 11
☒ 3.<(27)
=> true
☒ Pepita.==(Norita)
=> false
☒
```



Recapitulando

En un mundo de objetos, todo lo que tenemos son **objetos y mensajes**. A estos últimos, podemos distinguirlos según la forma en que se escriben:

☒ **Mensajes de palabra clave.** Su nombre está compuesto por una o varias palabras, puede terminar con un signo de exclamación ! o de pregunta ?, y se envía mediante un punto. Además,

- pueden no tomar argumentos, como Rayuela.anio_de_edicion ;
- o pueden tomar uno o más argumentos, separados por coma: SanMartin.cruzar!(LosAndes, Mula) .

☒ **Operadores.** Son todos aquellos cuyo "nombre" se compone de uno o más símbolos, y se envían simplemente escribiendo dichos símbolos.

En cuanto a los argumentos,

- pueden no tomar ninguno, como la negación !true ;
- o pueden tomar uno (y solo uno), como Orson == Garfield o energia + 80 .

Como vimos, también se pueden escribir como mensajes de palabra clave (aunque no parece buena idea escribir 1.==(2) en vez de 1 == 2 ☺).

Vamos a enviar algunos mensajes para terminar de cerrar la idea. Te toca escribir un programa que haga que Pepita:

1. Coma 90 gramos de alpiste. ☺
2. Vuelo a Iruya. ☺
3. Finalmente, coma tanto alpiste como el 10% de la energía que le haya quedado. ☺

Este programa tiene que andar sin importar con cuanta energía arranque Pepita .

☒ ¡Dame una pista!

Cualquier envío de mensajes que devuelva algo es una expresión válida, y puede ser usada en cualquier lugar en que se espera un objeto. Por ejemplo, las siguientes colaboraciones son válidas:

```
☒ Fitito.cargar_nafta!(120 * 4)
☒ Fitito.cargar_nafta!(Fitito.capacidad_tanque_nafta - Fitito.nafta_disponible) #Carga al Fitito lo necesario para completar su tanque. Para ello le pregunta al Fitito su capacidad y la nafta que tiene en este momento.
```

☒ **Solución** ☒ **Consola**

```
1 Pepita.comer_alpiste!(90)
2 Pepita.volar_hacia!(Iruya)
3 Pepita.comer_alpiste!(Pepita.energia / 10)
```

☒ 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 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)





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 declarar 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 **declaración** 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`.



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!

Para que no tengas que volver a escribirla, pusimos por vos la definición de Pepita en la **Biblioteca**. 📜

Siempre que te lo indiquemos, podrás usar los objetos de la Biblioteca como si los hubieras creado vos.

☒ Consola ☒ Biblioteca

```
☒ Pepita.energia
undefined method `energia' for Pepita:Module (NoMethodError)
☒ Pepita.comer_lombriz!
undefined method `comer_lombriz!' for Pepita:Module (NoMethodError)
☒ Pepita.volar_en_circulos!
undefined method `volar_en_circulos!' for Pepita:Module (NoMethodError)
☒
```



Los mejores, los únicos, los métodos en objetos

d_¿Otra vez undefined method? ¿Y ahora qué falta?_ 😞

Para que un objeto entienda un mensaje debemos "enseñarle" cómo hacerlo, y para ello es necesario declarar 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!

No te olvides de que el `!` forma parte del nombre del mensaje y por lo tanto tenés que escribirlo. 😊

Recordá también que podés usar la consola para ir probando tu solución.

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



Perdiendo energía

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

```
✗ Pepita.volar_en_circulos!
=> 90
✗ Pepita.volar_en_circulos!
=> 80
✗ Pepita.energia
undefined method `energia' for Pepita:Module (NoMethodError)
✗
```



Atributos

Analicemos el código que acabamos de escribir:

```
module Pepita
  @energia = 100

  def self.volar_en_circulos!
    @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_circulos!`, su energía disminuye: se realiza una nueva **asignación** del atributo y pasa a valer lo que valía antes (o sea, `@energia`), menos `10`.

Sabiendo esto, implementá 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_circulos!
5     @energia = @energia - 10
6   end
7
8   def self.comer_lombriz!
9     @energia = @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 `@`), que no son otra cosa que **referencias** a otros objetos.

Entonces, si `energia` es una referencia a un objeto, ¿los números también son objetos? 😊

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



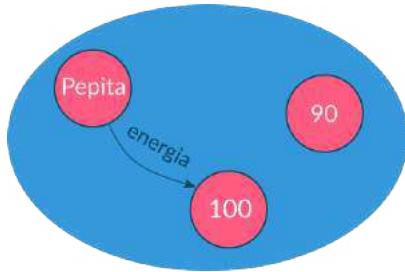
Asignaciones y referencias

Miremos este método con más detenimiento:

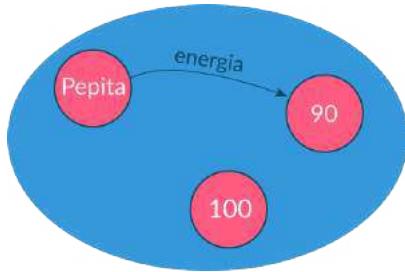
```
def volar_en_circulos!
    @energia = @energia - 10
end
```

Lo que estamos haciendo es cambiar la energía de `Pepita`: pasa de su valor actual, `@energia`, a ese valor menos `10`. Por ejemplo, pasa de `100` a `90`. ¿Significa esto que el `100` se transforma en un `90`? 😊

No, en absoluto. En objetos trabajamos con **referencias**: `energia` (un atributo) es una referencia a un objeto, que inicialmente apunta al objeto `100`. Si pensamos a los objetos como círculos y las referencias como flechas, podemos graficarlo de la siguiente manera:



Luego, la operación de asignación cambia ese apuntador, que pasa a referenciar al `90`:



En este caso se da una particularidad: el objeto asignado a la referencia es el resultado de **enviar el mensaje** - al objeto apuntado originalmente por la referencia: `@energia = @energia - 10`. Y como esta operación es tan común, se puede escribir de una forma más corta: `@energia -= 10`.

Reesribí los métodos que hiciste en el ejercicio anterior para que usen cuando puedan el `-=`, y su contrapartida, el `+=`.

[Solución](#) [Consola](#)

```
1 module Pepita
2     @energia = 100
3
4     def self.volar_en_circulos!
5         @energia -= 10
6     end
7
8     def self.comer_lombriz!
9         @energia += 20
10    end
11 end
```

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Esta guía fue desarrollada por Federico Aloï, Franco Bulgarelli, Ariel Umansky bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





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!](#)

Al parámetro de `volar_hacia!` tenés que darle un nombre. Podrías llamarlo `ciudad` , pero eso colisionaría con el nombre del atributo `ciudad` . Así que te proponemos otros nombres: `una_ciudad` o, mejor, `destino` ;

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

```
1 module Pepita
2   @energia = 100
3
4   def self.volar_en_circulos!
5     @energia -= 10
6   end
7
8   def self.comer_lombriz!
9     @energia += 20
10  end
11
12  @ciudad = Iruya
13
14  def self.volar_hacia!(destino)
15    @ciudad = destino
16    @energia -= 100
17  end
18
19 end
```

[✖ Enviar](#)

[✖ ¡Muy bien! Tu solución pasó todas las pruebas](#)



Leyendo el estado

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

```
✗ 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`? Declarando 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?

```
✗ Pepita.energia
=> 100
✗ Pepita.energia = 120
undefined method `energia=' for Pepita:Module (NoMethodError)
Did you mean? energia
✗ energia
undefined local variable or method `energia' for main:Object (NameError)
✗
```

Esta guía fue desarrollada por Federico Aloí, Franco Bulgarelli, Ariel Umansky bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





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 :

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

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

objetos)

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

Solución

Biblioteca

Consola



```
module Obera
  #...más cosas que ahora no interesan...
end

module Pepita
  @energia = 100
  @ciudad = Obera

  #...más cosas que ahora no interesan...
end

module Kiano1100
  #...más cosas que ahora no interesan...
end

module RolamotoC115
  #...más cosas que ahora no interesan...
end

module Enrique
  @celular = Kiano1100
  @dinero_en_billetera = 13
  @frase_favorita = 'la juventud está perdida'
end
```



¿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.cantar!
10    'pri pri pri'
11  end
12
13  def self.comer_lombriz!
14    @energia += 20
15  end
16
17  def self.volar_en_circulos!
18    @energia -= 10
19  end
20
21  def self.volar_hacia!(destino)
22    @energia -= 100
23    @ciudad = destino
24  end
25
26  def self.ciudad
27    @ciudad
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](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





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.

Si por ejemplo la distancia entre dos ciudades fuese de 1200 kilómetros, Pepita necesitaría 600 unidades de energía para llegar.

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



Sabiendo esto:

- Creá el objeto que representa a BuenosAires .
- Agregá a Obera , Iruya y BuenosAires un mensaje kilometro que devuelva la altura a la que se encuentran, según el esquema. ¡Ojo! No tenés que guardar el valor en un atributo @kilometro sino simplemente devolver el número que corresponde.
- Modificá el método volar_hacia! de Pepita la lógica necesaria para hacer el cálculo y alterar la 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!](#)

La distancia entre dos ciudades se puede calcular fácilmente restando sus kilómetros, peeeero...

...pensá que la energía que consume volar a BuenosAires desde Iruya tiene que ser la misma que para volar desde Iruya hasta BuenosAires , y en ambos casos tiene que ser positiva.

Acá te puede ser útil el mensaje abs que entienden los números :

```

☒ 17.abs
=> 17

☒ (-17).abs
=> 17

☒ (8 - 3).abs
=> 5

☒ (3 - 8).abs
=> 5

```

¡Podés probarlo en la consola!

[Solución](#) [Consola](#)

```

1 module BuenosAires
2   def self.kilometro
3     0
4   end
5 end
6
7 module Obera
8   def self.kilometro
9     1040
10  end
11 end

```

```

12
13 module Iruya
14   def self.kilometro
15     1710
16   end
17 end
18
19 module Pepita
20   @energia = 1000
21   @ciudad = Obera
22
23   def self.energia
24     @energia
25   end
26
27   def self.ciudad
28     @ciudad
29   end
30
31   def self.cantar!
32     'pri pri pri'
33   end
34
35   def self.comer_lombriz!
36     @energia += 20
37   end
38
39   def self.volar_en_circulos!
40     @energia -= 10
41   end
42
43   def self.volar_hacia!(destino)
44     @energia -= (destino.kilometro - @ciudad.kilometro).abs / 2
45     @ciudad = destino
46   end
47 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 Aloí, Franco Bulgarelli, Ariel Umansky bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





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 `distanzia_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_circulos!
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.distancia_a(destino)
31   (@ciudad.kilometro - destino.kilometro).abs
32 end
33
34 def self.gastar_energia!(destino)
35   @energia -= distancia_a(destino) / 2
36 end
37
38 end

```

► Enviar

 ¡Muy bien! Tu solución pasó todas las pruebas

La **delegación** es la forma que tenemos en objetos de **dividir en subtareas**: 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](https://creativecommons.org/licenses/by-sa/4.0/). (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)



```
module Obera
  def self.kilometro
    1040
  end
end

module Iruya
  def self.kilometro
    1710
  end
end

module BuenosAires
  def self.kilometro
    0
  end
end
```



¿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? ¿Cuál 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!](#)

Con las herramientas que vimos hasta ahora, no queda más opción que repetir el mismo código en las tres ciudades. 😬

¡Muy pronto lo solucionaremos!

Solución Consola

```
1 module Obera
2   def self.kilometro
3     1040
4   end
5   def self.distancia_a(destino)
6     (destino.kilometro - self.kilometro).abs
7   end
8 end
9
10 module Iruya
11   def self.kilometro
12     1710
13   end
14   def self.distancia_a(destino)
15     (destino.kilometro - self.kilometro).abs
16   end
17 end
18
19 module BuenosAires
20   def self.kilometro
21     0
22   end
23   def self.distancia_a(destino)
24     (destino.kilometro - self.kilometro).abs
25   end
26 end
27
28 module Pepita
29   @energia = 1000
30   @ciudad = Obera
31
32   def self.energia
33     @energia
34   end
35
36   def self.ciudad
37     @ciudad
```

```
38 end
39
40 def self.cantar!
41   'pri pri pri'
42 end
43
44 def self.comer_lombriz!
45   @energia += 20
46 end
47
48 def self.volar_en_circulos!
49   @energia -= 10
50 end
51
52 def self.volar_hacia!(destino)
53   self.gastar_energia!(destino)
54   @ciudad = destino
55 end
56
57 def self.gastar_energia!(destino)
58   @energia -= @ciudad.distancia_a(destino) / 2
59 end
60
61 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](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





¿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!

Recordá que existen los operadores de comparación `<` y `>`, que sirven para verificar si una expresión numérica es menor o mayor a otra, respectivamente.

Por ejemplo:

```
☒ 130 < 20 * 10
=> true
☒ Semana.cantidad_de_dias > 5
=> true
☒ 3 > PerroBoby.cantidad_de_patas
=> false
```

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 inventes tus propios mensajes, acordate que una de las funciones del código es **comunicar** nuestras ideas a otras personas... y las convenciones, muchas veces, nos ayudan con esto. 😊

Esta guía fue desarrollada por Federico Aloi y [muchas personas más](https://raw.githubusercontent.com/mumukiproject/mumuki-guia-ruby-polimorfismo/master/COLLABORATORS.txt) (<https://raw.githubusercontent.com/mumukiproject/mumuki-guia-ruby-polimorfismo/master/COLLABORATORS.txt>), bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





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:

```
def self.acomodar_habitacion!
  self.ordenar!
  if self.tiene_sabanas_sucias?
    self.cambiar_sabanas!
  end
  self.tender_la_cama!
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_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    end
28  end
29 end
30
31
```

✓ ¡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.



Esta guía fue desarrollada por Federico Aloi y muchas personas más (<https://raw.githubusercontent.com/mumukiproject/mumuki-guia-ruby-polimorfismo/master/COLLABORATORS.txt>), bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  Mumuki (<http://mumuki.org/>)





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`! ☺

```
def self.cuidar! (planta)
  if planta.necesita_agua?
    3.times { self.regar! (planta) }
  else
    self.sacar_bichos! (planta)
  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 por parámetro.

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

Modificá el código del ejercicio anterior 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 end
32
33
34
35
```

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Esta guía fue desarrollada por Federico Aloi y [muchas personas más](https://raw.githubusercontent.com/mumukiproject/mumuki-guia-ruby-polimorfismo/master/COLLABORATORS.txt) (<https://raw.githubusercontent.com/mumukiproject/mumuki-guia-ruby-polimorfismo/master/COLLABORATORS.txt>), bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





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 código:

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

34
35
36
37
38
39
40
41

► 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 (<https://raw.githubusercontent.com/mumukiproject/mumuki-guia-ruby-polimorfismo/master/COLLABORATORS.txt>), bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)





Llegó Pepo

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

- **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.

Implementá a Pepo según las reglas anteriores. Te dejamos el código de Pepita para usar como base, modificalo y borrar las partes que no correspondan.

[¡Dame una pista!](#)

En la Biblioteca te dejamos a Pepita por si no recordás cómo era la sintaxis de la alternativa condicional. [\(?\)](#)

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

```
1 module Pepo
2   @energia = 1000
3
4   def self.energia
5     @energia
6   end
7   def self.comer_alpiste!(gramos)
8     @energia += gramos / 2
9   end
10  def self.volar_en_circulos!
11    if self.energia > 1100
12      @energia -= 15
13    else
14      @energia -= 5
15    end
16  end
17  def self.hacer_lo_que_quiera!
18    self.comer_alpiste!(120)
19  end
20
21 end
```

[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_círculos!` y `hacer_lo_que_quiera!`.

Veamos qué podemos hacer con ellas...



¡A entrenar!

Nuestras aves quieren presentarse a las próximas Olimpiadas, y para eso necesitan ejercitarse 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!](#)

Recordá que la que entrena es Pepita, su entrenador solo le dice lo que tiene que hacer.

¡Ah! Y para que un objeto pueda mandarle mensajes a otro debe **conocerlo**. Por ejemplo, llamándolo por su nombre como ya hemos hecho:

Pepita.energia

[☰ 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
```

[☰ Enviar](#)

[☰ ¡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!



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 la referencia del ave de 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 mensaje `firmar_contrato!(ave)`, de forma tal que cuando le envíemos el mensaje `entrenar_ave!` haga entrenar al último ave con el que haya firmado contrato.

[¡Dame una pista!](#)

El método `firmar_contrato!` solo cambia el ave de Pachorra, no hay que hacer nada más en ese método. ☺

[Solución](#) [Consola](#)

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



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!
```

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



Un entrenamiento más duro

Analicemos el error:

```
☒ 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!

Para que `Emilce` pueda entrenar a un ave debe firmar un contrato tal como hace `Pachorra`.

☒ 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.



¿¿Polimor-qué??

¿Qué pasa si dos objetos, como Pepita , Norita o Pepo son capaces de responder a un mismo mensaje? Podemos cambiar la **referencia** de un objeto a 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.

¡Comprobemos si entendiste! Elegí las opciones correctas:

- Pepita, Norita y Pepo son polimórficas para Emilce.
- Pepita, Norita y Pepo son polimórficas para Pachorra.
- Pepita, Norita y Pepo no son polimórficas para Pachorra.
- Pepita y Pepo son polimórficas para Pachorra.

Enviar

¡La respuesta es correcta!

Para que quede clarísimo, una definición para leer todas las mañanas antes de desayunar:

Dos objetos son **polimórficos para un tercer objeto** cuando este puede enviarles los mismos **mensajes**, sin importar cómo respondan o qué otros mensajes entiendan.



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?



¡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 `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 end
19
```

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.



Cambiando la referencia

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 implementando el mensaje `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_circulos! }
    @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_circulos! }
8     @ave.comer_alpiste! 30
9     5.times { @ave.volar_en_circulos! }
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](https://raw.githubusercontent.com/mumukiproject/mumuki-guia-ruby-polimorfismo/master/COLLABORATORS.txt) (<https://raw.githubusercontent.com/mumukiproject/mumuki-guia-ruby-polimorfismo/master/COLLABORATORS.txt>), bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





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 agregaramos 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` ([https://es.wikipedia.org/wiki/Manuelita_\(canci%C3%B3n\)](https://es.wikipedia.org/wiki/Manuelita_(canci%C3%B3n))), una tortuga viajera. Algunos de sus atributos pueden ser leídos, otros modificados y otros ambas cosas.

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

☒ [¡Dame una pista!](#)

Recordá la convención para nombrar los métodos de acceso que mencionamos antes:

- Para los **getters**, que sirven para **obtener** el valor de un atributo, usamos el mismo nombre que este.
- Para los **setters**, que sirven para **fijar** el valor de un atributo, usamos el mismo nombre que este pero con un `=` al final.

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

```
1 atributos = %w(
2   energia
3   ciudad
4   mineral_preferido
5   donde_va
6 )
7
8 atributos_con_getter = %w(
9   energia
10  ciudad
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. 😊

```

module Obera
  def self.kilometro
    1040
  end

  def self.distancia_a(destino)
    (destino.kilometro - self.kilometro).abs
  end
end

module Iruya
  def self.kilometro
    1710
  end

  def self.distancia_a(destino)
    (destino.kilometro - self.kilometro).abs
  end
end

module BuenosAires
  def self.kilometro
    0
  end

  def self.distancia_a(destino)
    (destino.kilometro - self.kilometro).abs
  end
end

module Pehuajo
end

module Malaquita
end

module Paris
end

module Manuelita
  @energia = 100
  @ciudad = Pehuajo
  @mineral_preferido = Malaquita
  @donde_va = Paris

  def self.energia
    @energia
  end

  def self.ciudad
    @ciudad
  end

  def self.mineral_preferido=(mineral)
    @mineral_preferido = mineral
  end

  def self.mineral_preferido
    @mineral_preferido
  end

  def self.donde_va=(ciudad)
    @donde_va = ciudad
  end
end

```



Vamos terminando

Vamos a empezar a repasar todo lo que aprendiste en esta lección, te vamos a pedir que modeles a nuestro amigo **Inodoro** (https://es.wikipedia.org/wiki/Inodoro_Pereyra), un gaucho solitario de la pampa argentina. Fiel al estereotipo, Inodoro se la pasa tomando **mate** ([https://es.wikipedia.org/wiki/Mate_\(infus%C3%B3n\)](https://es.wikipedia.org/wiki/Mate_(infus%C3%B3n))), 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, implementando 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!](#)

Pará, pará, pará, ¿y cómo sé qué nombre le tengo que poner a los métodos? 😬

Como ya te explicamos, desde ahora nos vamos a manejar siempre con la misma convención para nombrar getters y setters. Podés buscarla en los ejercicios anteriores si aún no la recordás. ☺

☒ [Solución](#) ☒ [Consola](#)

```
1 module Inodoro
2   @cafeina_en_sangre = 90
3
4   def self.cafeina_en_sangre
5     @cafeina_en_sangre
6   end
7   def self.compinche
8     @compinche
9   end
10  def self.compinche=(nuevo_compinche)
11    @compinche = nuevo_compinche
12  end
13 end
14
15 module Eulogia
16   @enojada = false
17
18   def self.enojada?
19     @enojada
20   end
21 end
22
23 module Mendieta
24   @gananas_de_hablar = 5
25
26   def self.gananas_de_hablar
27     @gananas_de_hablar
28   end
29   def self.gananas_de_hablar=(gananas)
30     @gananas_de_hablar = gananas
31   end
```

► 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](#) (<https://raw.githubusercontent.com/mumukiproject/mumuki-guia-ruby-polimorfismo/master/COLLABORATORS.txt>), bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





¡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 `gananas_de_hablar` se vuelvan 0).

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

[☰ Dame una pista!](#)

Cuando `Inodoro` toma mate, su compinche recibe un mate:

```
☰ Inodoro.compinche= Mendieta
☰ Mendieta.ganas_de_hablar
=> 5
☰ Inodoro.tomar_mate!
☰ Mendieta.ganas_de_hablar
=> 0
```

Esto no funciona al revés, cuando sus compinches reciben un mate no lo hacen tomar a `Inodoro`:

```
☰ Inodoro.compinche= Eulogia
☰ Inodoro.cafeina_en_sangre
=> 90
☰ Eulogia.recibir_mate!
☰ Inodoro.cafeina_en_sangre
=> 90
```

[☰ Solución](#) [☰ Consola](#)

```
1 module Inodoro
2   @cafeina_en_sangre = 90
3
4   def self.cafeina_en_sangre
5     @cafeina_en_sangre
6   end
7   def self.compinche
8     @compinche
9   end
10  def self.compinche=(nuevo_compinche)
11    @compinche = nuevo_compinche
12  end
13  def self.tomar_mate!
14    @cafeina_en_sangre +=10
15    @compinche.recibir_mate!
16  end
17 end
18
19 module Eulogia
20   @enojada = false
21
22   def self.enojada?
23     @enojada
24   end
```

```
25 def self.recibir_mate!
26   @enojada = true
27 end
28 end
29
30 module Mendieta
31   @ganas_de_hablar = 5
32
33   def self.ganas_de_hablar
34     @ganas_de_hablar
35   end
36   def self.ganas_de_hablar=(ganas)
37     @ganas_de_hablar = ganas
38   end
39   def self.recibir_mate!
40     @ganas_de_hablar = 0
41   end
42 end
43
44
45
```

► Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Esta guía fue desarrollada por Federico Aloí y muchas personas más (<https://raw.githubusercontent.com/mumukiproject/mumuki-guia-ruby-polimorfismo/master/COLLABORATORS.txt>), bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)





Entrando en Calor

¡Vamos a crear una biblioteca de videojuegos! 🎮 Para empezar, tendremos tres videojuegos, de los cuales sabemos lo siguiente:

- 🎮 CarlosDuty : es violento. Su dificultad se calcula como `30 - @cantidad_logros * 0.5`. Y si se lo juega por más de 2 horas seguidas, se le suma un logro a su cantidad. Inicialmente, el juego no tiene logros.
- 🦁 TimbaElLeon : no es violento. Su dificultad inicial es 25 y crece un punto por cada hora que se juegue.
- 👾 Metroide : es violento sólo si `nivel_espacial` es mayor a 5. Este nivel arranca en 3 pero se incrementa en 1 cada vez que se lo juega, sin importar por cuánto tiempo. Además, su dificultad siempre es 100.

Declará estos tres objetos de forma que entiendan los mensajes `dificultad`, `violento?` y `jugar!(un_tiempo)`.

💡 Dame una pista!

- `un_tiempo` es una cantidad en horas;
 - tené en cuenta que si bien todos los juegos tienen que entender el mensaje `dificultad`, sólo `TimbaElLeon` tiene que tener un atributo `@dificultad`, los otros dos juegos no lo necesitan para decirnosla;
 - prestá atención a la dificultad de `TimbaElLeon`: La dificultad crece **tantas horas como se lo juegue**. Por ejemplo, si se lo juega durante **cuatro** horas, su dificultad se incrementa en **cuatro** puntos. Si se lo juega durante **siete** horas, su dificultad se incrementa en **siete** puntos.
- ¿Notás alguna coincidencia? 😊

✖ Solución ✖ Consola

```

1 module CarlosDuty
2   @cantidad_logros = 0
3
4   def self.violento?
5     @violento = true
6   end
7   def self.dificultad
8     @dificultad = 30 - (@cantidad_logros * 0.5)
9   end
10
11  def self.jugar!(un_tiempo)
12    if un_tiempo >= 2
13      @cantidad_logros += 1
14    end
15  end
16
17 end
18
19 module TimbaElLeon
20   @dificultad = 25
21
22   def self.violento?
23     @violento = false
24   end
25   def self.dificultad
26     @dificultad
27   end
28
29   def self.jugar!(un_tiempo)
30     @dificultad += un_tiempo
31   end
32 end
33
34 module Metroide

```

```
35 @nivel_espacial = 3
36
37 def self.dificultad
38   @dificultad = 100
39 end
40
41 def self.violento?
42   @violento = false
43   if @nivel_espacial > 5
44     @violento = true
45   end
46 end
47
48 def self.jugar!(un_tiempo)
49   @nivel_espacial += un_tiempo
50 end
51
52 end
53
```

► Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

¡Ya tenemos creados los objetos para nuestra colección de videojuegos! 🎮

Es importante que notes que todos estos objetos responden a los mismos mensajes: `dificultad`, `violento?` y `jugar!(un_tiempo)`. Como aprendiste con las golondrinas (./exercises/mumukiproject/mumuki-guia-ruby-polimorfismo/8), nuestros videojuegos son **polimórficos** para ese conjunto de mensajes.

¡Esto significa que podemos enviarles los mismos mensajes a cualquiera de los videojuegos y usarlos indistintamente! 🎮

Esta guía fue desarrollada por Felipe Calvo, Franco Bulgarelli, Mariana Matos, Gustavo Crespi bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  Mumuki (<http://mumuki.org/>)





Creando una lista

Ahora que ya tenemos nuestros videojuegos 🎮, vamos a ordenarlos en algún lugar.

Para ello necesitamos crear un objeto, la Biblioteca, que contenga otros objetos: nuestros videojuegos. Para ello vamos a usar una *lista* de objetos: es un tipo de colección en la cual los elementos pueden repetirse. Es decir, el mismo objeto puede aparecer más de una vez.

Por ejemplo, la lista de números 2, 3, 3 y 9 se escribe así:

```
[2, 3, 3, 9]
```

Veamos si se entiende: creá un objeto `Biblioteca` que tenga un atributo `juegos` con su correspondiente getter. La `Biblioteca` tiene que tener en primer lugar el juego `CarlosDuty`, luego `TimbaElLeon` y por último `Metroide`.

[.¡Dame una pista!](#)

⚠ ¡Cuidado con el orden! La lista es un tipo de colección donde el orden importa. Por lo tanto, no es lo mismo poner un objeto al principio que agregarlo al final. ¡Atención a eso! 😊

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

```
1 module Biblioteca
2 @juegos = [CarlosDuty, TimbaElLeon, Metroide]
3
4 def self.juegos
5   @juegos
6 end
7
8 end
```

[Enviar](#)

[¡Muy bien! Tu solución pasó todas las pruebas](#)

¡Excelente! Ya tenemos creada la `Biblioteca` con algunos juegos. 🎮

¿Pero qué más podemos hacer con las colecciones? Pasemos al siguiente ejercicio...

```
module CarlosDuty
  @cantidad_logros = 0

  def self.violento?
    @violento = true
  end
  def self.dificultad
    @dificultad = 30 - (@cantidad_logros * 0.5)
  end

  def self.jugar!(un_tiempo)
    if un_tiempo >= 2
      @cantidad_logros += 1
    end
  end

end

module TimbaElLeon
  @dificultad = 25

  def self.violento?
    @violento = false
  end
  def self.dificultad
    @dificultad
  end

  def self.jugar!(un_tiempo)
    @dificultad += un_tiempo
  end
end

module Metroide
  @nivel_espacial = 3

  def self.dificultad
    @dificultad = 100
  end

  def self.violento?
    @violento = false
    if @nivel_espacial > 5
      @violento = true
    end
  end

  def self.jugar!(un_tiempo)
    @nivel_espacial += un_tiempo
  end
end
```



Algunos mensajes básicos

¡Tengo una colección! ¿Y ahora qué...? 🤔

Todas las colecciones entienden una serie de mensajes que representan operaciones o consultas básicas sobre la colección.

Por ejemplo, podemos agregar un elemento enviándole `push` a la colección o quitarlo enviándole `delete`:

```
numeros_de_la_suerte = [6, 7, 42]
numeros_de_la_suerte.push 9
  # Agrega el 9 a la lista...
numeros_de_la_suerte.delete 7
  # ...y quita el 7.
```

También podemos saber si un elemento está en la colección usando `include?`:

```
numeros_de_la_suerte.include? 6
  # Devuelve true, porque contiene al 6...
numeros_de_la_suerte.include? 8
  # ...devuelve false, porque no contiene al 8.
```

Finalmente, podemos saber la cantidad de elementos que tiene enviando `size`:

```
numeros_de_la_suerte.size
  # Devuelve 3, porque contiene al 6, 42 y 9
```

¡Probá enviarle los mensajes `push`, `delete`, `include?` y `size` a la colección `numeros_de_la_suerte`!

💡 ¡Dame una pista!

Recordá que, además de los mensajes que vimos recién, podés enviar simplemente `numeros_de_la_suerte` en la consola para ver qué elementos componen a la colección. 😊

💡 Consola 💡 Biblioteca

```
numeros_de_la_suerte
=> [6, 42, 9]
numeros_de_la_suerte.delete 42
=> 42
numeros_de_la_suerte.push 12
=> [6, 9, 12]
numeros_de_la_suerte.size
=> 3
```



Mejorando la Biblioteca

Primero nos encargamos de los videojuegos, y ahora ya conocés qué mensajes entienden las listas. ¡Es momento de darle funcionalidad a la Biblioteca! 🎮

Nuestra Biblioteca maneja puntos. Agregá el código necesario para que entienda los siguientes mensajes:

- puntos : nos dice cuantos puntos tiene la Biblioteca. Inicialmente son 0.
- adquirir_juego!(un_juego) : agrega el juego a la Biblioteca , y le suma 150 puntos.
- borrar_juego!(un_juego) : quita un juego de la Biblioteca , pero no resta puntos.
- completa? : se cumple si la Biblioteca tiene más de 1000 puntos y más de 5 juegos.
- juego_recomendable?(un_juego) : es verdadero para un_juego si no está en la Biblioteca y es violento? .

☒ ¡Dame una pista!

Para saber si la Biblioteca es completa? necesitás que se cumplan dos condiciones a la vez: que tenga más de 1000 puntos y más de 5 juegos. Y para juego_recomendable? necesitás que el juego no esté en la Biblioteca . Quizá && y ! te sean útiles en estos dos casos:

```
☒ 7 > 3 && Mario.hermano_de?(Luigi)
=> true

☒ !Pepita.sabe_matematica?
=> true
```

☒ Solución ☒ Consola

```
1 module Biblioteca
2 @juegos = [CarlosDuty, TimbaElLeon, Metroide]
3 @puntos = 0
4
5 def self.juegos
6   @juegos
7 end
8
9 def self.puntos
10   @puntos
11 end
12
13 def self.adquirir_juego!(un_juego)
14   @juegos.push(un_juego)
15   @puntos+=150
16 end
17
18 def self.borrar_juego!(un_juego)
19   @juegos.delete(un_juego)
20 end
21
22 def self.completa?
23   @puntos > 1000 && @juegos.size > 5
24 end
25
26 def self.juego_recomendable?(un_juego)
27   !@juegos.include?(un_juego) && un_juego.violento?
28 end
29
30 end
```

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Hay una diferencia notable entre los primeros dos mensajes (`push` y `delete`) y los otros dos (`include?` y `size`):

1. `push` y `delete`, al ser evaluados, *modifican* la colección. Dicho de otra forma, producen un **efecto** sobre la lista en sí: agregan o quitan un elemento del conjunto.
2. `include?` y `size` sólo nos retornan información sobre la colección. Son métodos **sin efecto**.

Ahora que ya dominás las listas, es el turno de subir un nivel más... ☺

Esta guía fue desarrollada por Felipe Calvo, Franco Bulgarelli, Mariana Matos, Gustavo Crespi bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





¿Bloques? ¿Eso se come?

¡Pausa! 🖐️ Antes de continuar, necesitamos conocer a unos nuevos amigos: los *bloques*.

Los *bloques* son **objetos** que representan un mensaje o una secuencia de envíos de mensajes, **sin ejecutar**, lista para ser evaluada cuando corresponda. La palabra con la que se definen los bloques en Ruby es *proc*. Por ejemplo, en este caso le asignamos un *bloque* a *incrementador*:

```
un_numero = 7
incrementador = proc { un_numero = un_numero + 1 }
```

Ahora avancemos un pasito: en este segundo ejemplo, al *bloque* `{ otro_numero = otro_numero * 2 }` le enviamos el mensaje *call*, que le indica que **evalúe la secuencia de envíos de mensajes dentro de él**.

```
otro_numero = 5
duplicador = proc { otro_numero = otro_numero * 2 }.call
```

¡Es hora de poner a prueba tu conocimiento! 😊 Marcá las respuestas correctas:

☒ [.¡Dame una pista!](#)

¿Cuánto vale `un_numero` luego de las primeras dos líneas? Prestá atención a la explicación: la secuencia de envío de mensajes en el *bloque* del primer ejemplo está **sin ejecutar**. En cambio, al enviar el mensaje `call` en el ejemplo de `otro_numero` ... 😊

- `un_numero` vale 7
- `un_numero` vale 8
- `otro_numero` vale 5
- `otro_numero` vale 10

☒ Enviar

☒ ¡La respuesta es correcta!

¡Muy bien! Repasemos entonces:

- `un_numero` vale 7, porque el bloque `incrementador` no está aplicado. Por tanto, no se le suma 1.
- `otro_numero` vale 10, porque el bloque `duplicador` se aplica mediante el envío de mensaje `call`, que hace que se ejecute el código dentro del bloque. Por tanto, se duplica su valor.



Bloques con parámetros

Los bloques también pueden recibir parámetros para su aplicación. Por ejemplo, `sumar_a_otros_dos` recibe dos parámetros, escritos entre barras verticales `|` y separados por comas:

```
un_numero = 3
sumar_a_otros_dos = proc { |un_sumando, otro_sumando| un_numero = un_numero + un_sumando + otro_sumando }
```

Para aplicar el bloque `sumar_a_otros_dos`, se le pasan los parámetros deseados al mensaje `call`:

```
sumar_a_otros_dos.call(1,2)
=> 6
```

Volvamos a los videojuegos... Asignale a la variable `jugar_a_timba` un bloque que reciba un único parámetro. El bloque recibe una cantidad de minutos y debe hacer que se juegue a `TimbaElLeon` durante ese tiempo, pero recordá que `jugar!` espera una cantidad de horas. ¡Podés ver el objeto `TimbaElLeon` en la solapa Biblioteca (☞)!

¡Dame una pista!

- Para pasar de minutos a horas simplemente tenés que dividir esa cantidad de minutos por 60. Por ejemplo:

```
120/60
=> 2 #Porque 120 minutos son dos horas
```

- ¿Y cómo se hace en los casos en los que el bloque recibe un único parámetro, en lugar de dos? ¡Fácil! ☠ Se escribe de la misma forma, entre barras verticales `|`, sin utilizar comas.

Solución Biblioteca Consola

```
1 jugar_a_timba = proc { |tiempo_en_minutos| TimbaElLeon.jugar!(tiempo_en_minutos/60) }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Quizá estés pensando, *¿qué tiene que ver todo esto con las colecciones?* ;Paciencia! ☹ En el siguiente ejercicio veremos cómo combinar colecciones y bloques para poder enviar mensajes más complejos.



Filtrando quienes cumplen

¿Qué pasa cuando queremos todos aquellos objetos que cumplan con una condición determinada en una cierta colección? Por ejemplo, si de una lista de números queremos los mayores a 3.

Lo que usamos es el mensaje `select` de las colecciones. `select` recibe un *bloque* con un parámetro que representa un elemento de la colección y una condición booleana como código, y lo que devuelve es una nueva colección con los elementos que la cumplen.

```
algunos_numeros = [1, 2, 3, 4, 5]
mayores_a_3 = algunos_numeros.select { |un_numero| un_numero > 3 }
```

¿Y cuándo se aplica ese bloque que recibe el `select`? ¡El `select` es quien decide! 🤔 La colección va a aplicarlo con cada uno de los objetos (`un_numero`) cuando corresponda durante el seleccionado (o filtrado) de elementos.

```
mayores_a_3
=> [4, 5]
```

Mientras tanto, en nuestra biblioteca de videojuegos...

¡Ahora te toca a vos! Agregá el método `juegos_violentos` que retorna los juegos de la Biblioteca que cumplan violento? .

☰ ¡Dame una pista!

¿A qué se parece el `select` de objetos? ¡Sí! ¡Al `filter` de funcional (<https://central.mumuki.io/exercises/1726-programacion-funcional-listas-cantidadtuitscortos>)!

☰ Solución ☰ Biblioteca ☰ Consola

```
1 module Biblioteca
2 @juegos = [CarlosDuty, TimbaElLeon, Metroide]
3
4 def self.juegos
5   @juegos
6 end
7
8 def self.juegos_violentos
9   juegos_violentos = @juegos.select { |juegos_violentos| juegos_violentos.violento? }
10 end
11 end
```

☰ Enviar

☰ ¡Muy bien! Tu solución pasó todas las pruebas

¿Y qué pasa con la colección original, como `algunos_numeros` o `juegos`? ¿Se modifica al aplicar `select`? 🤔

¡No, para nada! El `select` no produce efecto.



El que busca encuentra

¿Y si en vez de **todos** los elementos que cumplan una condición, sólo queremos uno? ¡Usamos `find`!

```
algunos_numeros = [1, 2, 3, 4, 5]
uno_mayor_a_3 = algunos_numeros.find { |un_numero| un_numero > 3 }
```

Mientras que `select` devuelve una colección, `find` devuelve **únicamente** un elemento.

```
✗ uno_mayor_a_3
=> 4
```

¿Y si ningún elemento de la colección cumple la condición? Devuelve `nil`, que, como aprendiste antes, es un objeto que representa la nada - o en este caso, que ninguno cumple la condición. 😊

Veamos si se entiende: hacé que la biblioteca entienda `juego_mas_dificil_que(una_dificultad)`, que retorna algún juego en la biblioteca con más dificultad que la que se pasa por parámetro.

✗ Solución ✗ Biblioteca ✗ Consola

```
1 module Biblioteca
2 @juegos = [CarlosDuty, TimbaElLeon, Metroide]
3
4 def self.juegos
5   @juegos
6 end
7
8 def self.juego_mas_dificil_que(una_dificultad)
9   juego_mas_dificil = @juegos.find { |juego_mas_dificil| juego_mas_dificil.dificultad > una_dificultad}
10 end
11 end
```

✗ Enviar

✗ ¡Muy bien! Tu solución pasó todas las pruebas

Un dato curioso para tener en cuenta: ¡los mensajes `find` y `detect` hacen exactamente lo mismo! 😊



¿Alguno cumple? ¿Todos cumplen?

Para saber si **todos** los elementos de una colección cumplen un cierto criterio podemos usar el mensaje `all?`, que también recibe un bloque. Por ejemplo, si tenemos una colección de alumnos, podemos saber si todos aprobaron 😊 de la siguiente forma:

```
alumnos.all? { |un_alumno| un_alumno.aprobo? }
```

De manera muy similar podemos saber si **alguno** de la colección cumple cierta condición mediante el mensaje `any?`. Siguiendo el ejemplo anterior, ahora queremos saber si por lo menos uno de nuestros alumnos aprobó 😎 :

```
alumnos.any? { |un_alumno| un_alumno.aprobo? }
```

```
def self.metodo?  
  colección.all?{|elemento| elemento.condicion?}  
end
```

Declará los siguientes métodos en nuestra `Biblioteca`:

- `muchas_violencias?` : se cumple si todos los juegos que posee son violentos.
- `muy_dificil?` : nos dice si alguno de los juegos tiene más de 25 puntos de dificultad.

☒ Solución ☒ Biblioteca ☒ Consola

```
1 module Biblioteca  
2 @juegos = [CarlosDuty, TimbaElLeon, Metroide]  
3  
4 def self.juegos  
5   @juegos  
6 end  
7  
8 def self.muchas_violencias?  
9   @juegos.all? { |juego| juego.violento? }  
10 end  
11  
12 def self.muy_dificil?  
13   @juegos.any? { |juego| juego.dificultad > 25}  
14 end  
15  
16 end
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas

¿Qué tienen de distinto `all?` y `any?` respecto a `select` y `find`?

Mientras que `select` devuelve una colección y `find` un elemento o `nil`, `all?` y `any?` siempre devuelven un valor booleano: `true` o `false`.

Siguiente Ejercicio: El viejo y querido map



El viejo y querido map

El mensaje `map` nos permite, a partir de una colección, obtener **otra colección** con cada uno de los resultados que retorna un envío de mensaje a cada elemento.

En otras palabras, la nueva colección tendrá lo que devuelve el mensaje que se le envíe a cada uno de los elementos. Por ejemplo, si usamos `map` para saber los niveles de energía de una colección de golondrinas:

```
[Pepita, Norita].map { |una_golondrina| una_golondrina.energia }
=> [77, 52]
```

Al igual que el resto de los mensajes que vimos hasta ahora, `map` no modifica la colección original ni sus elementos, sino que devuelve una **nueva colección**.

Agregá a la `Biblioteca` un método llamado `dificultad_violenta` que retorne una colección con la dificultad de sus `juegos_violentos`.

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

```
1 module Biblioteca
2 @juegos = [CarlosDuty, TimbaElLeon, Metroide]
3
4 def self.juegos
5   @juegos
6 end
7
8 def self.juegos_violentos
9   juegos_violentos = @juegos.select { |juegos_violentos| juegos_violentos.violento? }
10 end
11
12 def self.dificultad_violenta
13   juegos_violentos.map { |juego| juego.dificultad }
14 end
15
16 end
```

[Enviar](#)

¡Muy bien! Tu solución pasó todas las pruebas

Antes de seguir, un caso particular. Dijimos que `map` **no** modifica la colección original. Pero, ¿qué ocurriría si el mensaje dentro del bloque en el `map` **sí tiene efecto**?

En ese caso **se modificaría la colección original**, pero sería un **mal uso del `map`**. Lo que nos interesa al mapear es lo que devuelve el mensaje que enviamos, no provocar un efecto sobre los objetos.

```
module CarlosDuty
  @cantidad_logros = 0

  def self.violento?
    @violento = true
  end
  def self.dificultad
    @dificultad = 30 - (@cantidad_logros * 0.5)
  end

  def self.jugar!(un_tiempo)
    if un_tiempo >= 2
      @cantidad_logros += 1
    end
  end

end

module TimbaElLeon
  @dificultad = 25

  def self.violento?
    @violento = false
  end
  def self.dificultad
    @dificultad
  end

  def self.jugar!(un_tiempo)
    @dificultad += un_tiempo
  end
end

module Metroide
  @nivel_espacial = 3

  def self.dificultad
    @dificultad = 100
  end

  def self.violento?
    @violento = false
    if @nivel_espacial > 5
      @violento = true
    end
  end

  def self.jugar!(un_tiempo)
    @nivel_espacial += un_tiempo
  end
end
```



¿Cuántos cumplen? ¿Cuánto suman?

Volviendo a nuestra colección de alumnos. Ya preguntamos si todos aprobaron o si alguno aprobó utilizando `all?` y `any?`. ¿Y si queremos saber **cuántos** aprobaron? Usamos `count`:

```
alumnos.count { |un_alumno| un_alumno.aprobo? }
```

`count` nos dice cuántos elementos de una colección cumplen la condición. Por otro lado, para calcular sumatorias tenemos el mensaje `sum`. Si queremos conocer la suma de todas las notas de los alumnos, por ejemplo, podemos hacer:

```
alumnos.sum { |un_alumno| un_alumno.nota_en_examen }
```

Veamos si se entiende: agregá a la `Biblioteca` el método `promedio_de_violencia`, cuyo valor sea la sumatoria de dificultad de los juegos violentos dividida por la cantidad de juegos violentos de la `Biblioteca`.

¡Dame una pista!

¡Atención a la división en `promedio_de_violencia`! Te recomendamos pensarla en dos partes:

- Primero, necesitás la sumatoria de la dificultad de los juegos violentos.
- A ese valor, lo dividís por la cantidad de juegos violentos.

¡Recordá que podés partir un problema *delegando* en varios mensajes!

Solución Consola

```
1 module Biblioteca
2 @juegos = [CarlosDuty, TimbaElLeon, Metroide]
3
4 def self.juegos
5   @juegos
6 end
7
8 def self.juegos_violentos
9   juegos_violentos = @juegos.select { |juegos_violentos| juegos_violentos.violento? }
10 end
11
12 def self.promedio_de_violencia
13   self.juegos_violentos.sum { |juego| juego.dificultad } / self.juegos_violentos.count { |juego|
juego.violento? }
14 end
15
16 end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



Jugando a todo

Hasta ahora, todos los mensajes que vimos de colecciones (con la excepción de `push` y `delete`) no están pensados para producir efectos sobre el sistema. ¿Qué ocurre, entonces, cuando queremos *hacer* algo con cada elemento? A diferencia del `map`, no nos interesan los resultados de enviar el mismo mensaje a cada objeto, sino mandarle un mensaje a cada uno con la intención de **producir un efecto**.

Es en este caso que nos resulta de utilidad el mensaje `each`.

Por ejemplo, si queremos que de una colección de golondrinas, aquellas con energía mayor a 100 vuelen a Iruya, podríamos combinar `select` y `each` para hacer:

```
golondrinas
  .select { |una_golondrina| una_golondrina.energia > 100 }
  .each { |una_golondrina| una_golondrina.volar_hacia!(Iruya) }
```

Ya que casi terminamos la guía y aprovechando que tenemos una colección de videojuegos, lo que queremos es... ¡jugar a todos! 😊

Definí el método `jugar_a_todo!` en la `Biblioteca`, que haga jugar a cada uno de los juegos durante 5 horas. Recordá que los juegos entienden `jugar!` (`un_tiempo`).

Solución Consola

```
1 module Biblioteca
2 @juegos = [CarlosDuty, TimbaElLeon, Metroide]
3
4 def self.juegos
5   @juegos
6 end
7
8 def self.jugar_a_todo!
9   @juegos.each { |juego| juego.jugar!(5) }
10 end
11
12
13 end
14
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



Variables

Hasta ahora, en objetos, un programa es simplemente una secuencia de envíos de mensajes. Por ejemplo, éste es un programa que convierte en mayúsculas al string "hola".

```
☒ "hola".upcase  
=> "HOLA"
```

Sin embargo, podemos hacer algo más: declarar variables. Por ejemplo, podemos declarar una variable `saludo`, inicializarla con "hola", enviarle mensajes...

```
☒ saludo = "hola"  
☒ saludo.upcase  
=> "HOLA"
```

...y esperar el mismo resultado que para el programa anterior.

Veamos si queda claro: agregá al programa anterior una variable `saludo_formal`, inicializada con "buen día"

☒ Solución ☒ Consola

```
1 saludo = "hola"  
2 saludo_formal = "buen día"  
3
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas

¡Momento, momento! 🙌

¿Qué sucedió aquí? Hasta ahora habíamos visto que tenemos objetos y mensajes, y sólo le podíamos enviar mensajes a los objetos, como `Pepita, 14, u "hola"`. ¿Le acabamos de enviar un mensaje a una variable?

Sí y no. Veamos por qué...



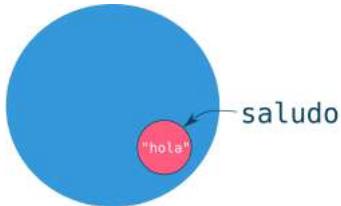
Las variables son referencias

Hasta ahora venimos insistiendo con que, en la programación en objetos, le enviamos mensajes a los objetos. ¡Y no mentimos!

Sucede que en realidad las cosas son un poco más complejas: no conocemos a los objetos directamente, sino a través de etiquetas llamadas *referencias*. Entonces cuando tenemos una declaración de variable como ésta...

```
saludo = "hola"
```

...lo que estamos haciendo es *crear una referencia* `saludo` que *apunta* al objeto "hola", que representamos mediante una flechita:



Y cuando tenemos...

```
saludo.uppercase
```

...le estamos enviando el mensaje `uppercase` al objeto "hola", a través de la referencia `saludo`, que es una variable.

Veamos si se entiende hasta acá: creá una variable llamada `despedida` que apunte al objeto "adiós", y luego enviale el mensaje `size()`.

☒ ¡Dame una pista!

¡No olvides que adiós va con tilde en la ó!

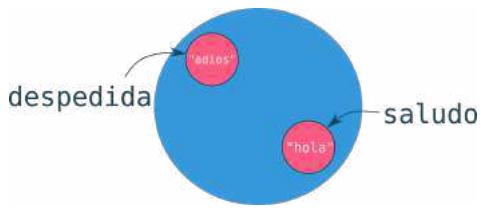
☒ Solución ☒ Consola

```
1 despedida = "adiós"  
2 despedida.size()
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas

¡Bien! Acabás de crear este **ambiente**, en criollo, el lugar donde viven los objetos con los cuales podemos interactuar:



También podemos hacer cosas como `"hola".size`. Allí no hay ninguna variable: ¿dónde está la referencia en ese caso? ¡Allá vamos!

Esta guía fue desarrollada por Felipe Calvo, Franco Bulgarelli bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Referencias implícitas

Como vemos, los objetos son las "bolitas" y las referencias, las "flechitas". Pero, ¿cuál es la diferencia entre variable y referencia?

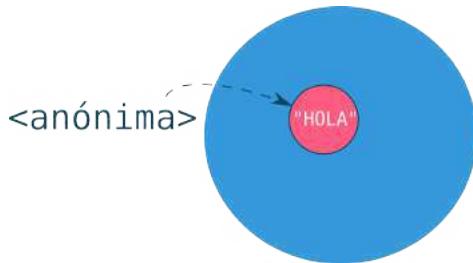
Sucede que hay muchos tipos de referencias, y una de ellas son las variables del programa. Pero, ¿no podíamos enviarles mensajes "directamente" al objeto? Por ejemplo, ¿dónde están las referencias en estos casos?:

```
#¿A qué referencia el envío upcase?  
"ni hao".upcase
```

```
#¿Y a qué referencia el envío size?  
saludo.upcase.size
```

;Simple! Cuando enviamos mensajes a objetos literales como el 2, el true u "hola", o expresiones, estamos conociendo a esos objetos a través de **referencias implícitas**, que son **temporales** (sólo existen durante ese envío de mensajes) y **anónimas** (no tienen un nombre asociado).

```
"ni hao".upcase  
^  
+-- Acá hay una referencia implícita al objeto "ni hao"  
  
saludo.upcase.size  
^  
+-- Y acá, otra referencia implícita a "HOLA"
```



Por eso, si luego te interesa hacer más cosas con ese objeto, tenés que crear una referencia explícita al mismo ⓘ. Las referencias explícitas son las que vimos hasta ahora. Por ejemplo:

```
saludoEnChino = "ni hao"
```

Probá las siguientes consultas en la consola y pensá en dónde hay referencias implícitas:

- ⓘ "ni hao".upcase
- ⓘ 4.abs.even?
- ⓘ (4 + 8).abs

```
 ⓘ "ni hao".upcase  
=> "NI HAO"  
 ⓘ 4.abs.even?  
=> true  
 ⓘ (4 + 8).abs  
=> 12  
 ⓘ
```



Múltiples referencias

Supongamos que tenemos el siguiente programa:

```
otro_saludo = "buen día"  
despedida = otro_saludo
```

Como vemos, estamos asignando `otro_saludo` a `despedida`. ¿Qué significa esto? ¿Acabamos de copiar el objeto "buen día", o más bien le dimos una nueva etiqueta al mismo objeto? Dicho de otra forma: ¿apuntan ambas variables al mismo objeto?

¡Averigualo vos mismo! Declará las variables `otro_saludo` y `despedida` como en el ejemplo de más arriba, y realizá las siguientes consultas:

- ☒ "buen día".equal? "buen día"
- ☒ despedida.equal? "buen día"
- ☒ otro_saludo.equal? otro_saludo
- ☒ despedida.equal? otro_saludo

¡Ahora sacá tus conclusiones! 😊

```
☒ "buen día".equal? "buen día"  
=> false  
☒ despedida.equal? "buen día"  
undefined local variable or method `despedida' for main:Object (NameError)  
☒ otro_saludo.equal? otro_saludo  
undefined local variable or method `otro_saludo' for main:Object (NameError)  
☒ despedida.equal? otro_saludo  
undefined local variable or method `despedida' for main:Object (NameError)
```



Identidad, revisada

Recordemos que el `equal?` era un mensaje que nos decía si dos objetos son el mismo. Veamos qué pasó:

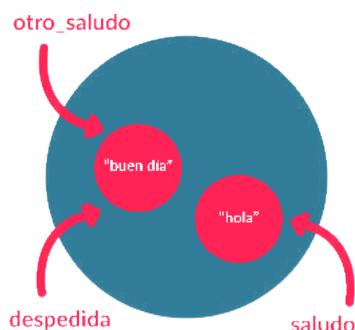
```
otro_saludo = "buen día" # se crea la variable otro_saludo que referencia al objeto "buen día"
despedida = otro_saludo # se crea la variable despedida que, por asignarle la referencia otro_saludo, apunta al mismo objeto
```

```
☒ "buen día".equal? "buen día"
=> false
☒ despedida.equal? "buen día"
=> false
```

En ambos casos el resultado fue `false`, dado que aquellos strings son objetos **distintos**, a pesar de que tengan los mismos caracteres. Cada vez que escribimos un string estamos creando un nuevo objeto. Sin embargo:

```
☒ otro_saludo.equal? otro_saludo
=> true
☒ despedida.equal? otro_saludo
=> true
```

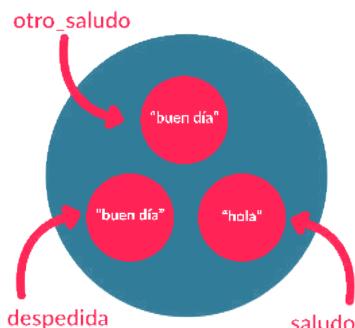
¿Por qué? ¡Simple! Ambas referencias, `otro_saludo` y `despedida`, apuntan al mismo objeto. La moraleja es que declarar una variable significa agregar una nueva referencia al objeto existente, en lugar de copiarlo:



Distinto sería si hacemos:

```
otro_saludo = "buen día"
despedida = "buen día"
```

Lo cual da como resultado este ambiente:



Veamos si se entiende: declará una lista `referencias_repetidas`, que esté conformada por tres referencias a un mismo objeto (¡el que quieras!)

Solución

> Consola

```
1 referencias_repetidas = [  
2 comida = "manzana",  
3 fruta = comida,  
4 vegetal = comida  
5 ]  
6  
7
```



► Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Ya entendimos que dos strings con el mismo contenido no necesariamente son el mismo objeto. Pero esto puede ser poco práctico 😞 . ¿Cómo hacemos si realmente queremos saber si dos objetos, pese a no ser el mismo, tienen el mismo estado?

Seguinos...

Esta guía fue desarrollada por Felipe Calvo, Franco Bulgarelli bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Equivalencia

Entonces, ¿qué pasa si lo que quiero es comparar los objetos no por su identidad, sino por que representen la misma cosa?

Pensemos un caso concreto. ¿Hay forma de saber si dos strings representan la misma secuencia de caracteres más allá de que no sean el mismo objeto? ¡Por supuesto que la hay! Y no debería sorprendernos a esta altura que se trate de otro mensaje:

```
☒ "hola" == "hola"  
=> true  
☒ "hola" == "adiós"  
=> false  
☒ "hola".equal? "hola"  
=> false
```

El mensaje `==` nos permite comparar dos objetos por *equivalencia*; lo cual se da típicamente cuando los objetos tienen el mismo estado. Y como vemos, puede devolver `true`, aún cuando los dos objetos no sean *el mismo*.

Veamos si se entiende: declará una variable `objetos_equivaleentes`, que referencie a una lista conformada por tres referencias **distintas** que apunten a objetos equivalentes entre sí, pero no idénticos.

☒ Solución ☒ Consola

```
1 objetos_equivaleentes = [  
2   mueble = "silla",  
3   otro_mueble = "silla",  
4   un_mueble_mas = "silla"  
5 ]
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas

⚠ ¡Ojo! A diferencia de la identidad, que todos los objetos la entienden sin tener que hacer nada especial, la equivalencia es un poco más complicada.

- Por defecto, si bien todos los objetos también la entienden, *delega* en la identidad, así que muchas veces es lo mismo enviar uno u otro mensaje
- Y para que realmente compare a los objetos por su estado, vos tenés que implementar este método a mano en cada objeto que crees. Los siguientes objetos ya la implementan:
 - Listas
 - Números
 - Strings
 - Booleanos



Objetos bien conocidos

¿Y qué hay de los objetos que veníamos declarando hasta ahora? Por ejemplo a `Fito`, le aumenta la felicidad cuando come:

```
module Fito
  @felicidad = 100

  def self.comer(calorías)
    @felicidad += calorías * 0.001
  end

  def self.felicidad
    @felicidad
  end
end
```

A objetos como `Fito` se los conocen como *objetos bien conocidos*: cuando los declaramos no sólo describimos su comportamiento (`comer(calorías)` y `felicidad`) y estado (`@felicidad`), sino que además les damos un nombre o etiqueta a través de la cual podemos conocerlos. ¿Te suena?

¡Adiviná! Esas etiquetas también son referencias . Y son globales, es decir que cualquier objeto o **programa** puede utilizarla.

Veamos si va quedando claro. Declará un objeto `AbuelaClotilde` que entienda un mensaje `alimentar_nieto`, que haga `comer` 2 veces a `Fito`: primero con 2000 calorías, y luego con 1000 calorías; ¡el postre no podía faltar! .

Solución Consola

```
1 #Ya declaramos a Fito por vos.
2 #¡Desarrollá a la AbuelaClotilde acá!
3 module AbuelaClotilde
4   def self.alimentar_nieto
5     Fito.comer(2000)
6     Fito.comer(1000)
7   end
8 end
9
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Muchas veces, en lugar de decir que le enviamos un mensaje al objeto apuntado por la referencia `Fito`, podemos llegar a decir...

enviar un mensaje a la variable `Fito`

...o...

enviar un mensaje al objeto Fito

...o simplemente...

enviar un mensaje a Fito

...porque si bien no es del todo correcto, es más breve 😊. Lo importante es que *siempre* estamos enviando el mensaje al objeto a través de una referencia.

Esta guía fue desarrollada por Felipe Calvo, Franco Bulgarelli bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Atributos y parámetros

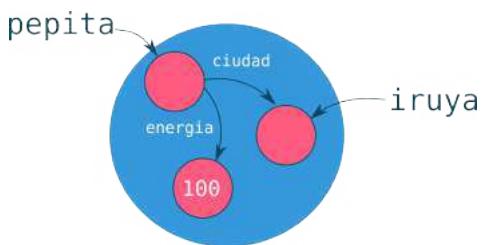
Además de los que ya vimos, hay más tipos de referencias: los atributos.

Por ejemplo, si la golondrina `Pepita` conoce siempre su ciudad actual...

```
module Pepita
  @ciudad

  def self.ciudad=(una_ciudad)
    @ciudad = una_ciudad
  end
end
```

Y en algún momento esta pasa a ser `Iruya`, el diagrama de objetos será el siguiente:



Nuevamente, acá vemos otro caso de múltiples referencias: el objeto que representa a la ciudad de `Iruya` (<https://es.wikipedia.org/wiki/Iruya>) es globalmente conocido como `Iruya`, y también conocido por `Pepita` como `ciudad`.

Escribí un programa que defina la `ciudad` de `Pepita` de forma que apunte a `Iruya`. Y pensá: ¿cuántas referencias a `Iruya` hay en este programa? ☰

Solución Consola

```
1 #Ya definimos a Pepita por vos.
2 #Ahora definí su ciudad...
3 Pepita.ciudad = Iruya
4
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¿Lo pensaste? 😊

Hay tres referencias a este objeto:

1. La propia referencia `Iruya`
2. El atributo `ciudad` de `Pepita`

3. una_ciudad : porque los parámetros de los métodos ¡también son referencias! Sólo que su vida es más corta: viven lo que dure la evaluación del método en el que se pasan.

Esta guía fue desarrollada por Felipe Calvo, Franco Bulgarelli bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Objetos compartidos

¿Te acordás de Fito? Fito también tiene una novia, Melisa. Melisa es nieta de AbueloGervasio. Cuando Melisa es feliz Fito es feliz:

```
module Fito
  @novio

  def self.novia=(una_novia)
    @novia = un_novia
  end

  def self.es_feliz_como_su_novia?
    @novia.felicidad > 105
  end
end
```

Escribí un programa que inicialice la novia de Fito y la nieta de AbueloGervasio de forma que ambos conozcan al mismo objeto (Melisa).

Luego, hacé que el abuelo alimente a su nieta 3 veces. ¿Qué pasará con Fito? ¿Se pondrá feliz?

¡Dame una pista!

¡Recordá que los números entienden el mensaje `times`, que recibe un bloque y lo ejecuta tantas veces como el valor del número!

También recordá que para que el abuelo alimente a su nieta tenés que enviarle el mensaje `alimentar_nieta`.

Solución Consola

```
1 #Melisa, Fito y AbueloGervasio ya están declarados.
2 #Inicializalos y enviales mensajes acá...
3 Fito.novia = Melisa
4 AbueloGervasio.nieta = Melisa
5
6 3.times { AbueloGervasio.alimentar_nieta }
7
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

En el programa que acabás de escribir, que probablemente se vea parecido a esto...

```
Fito.novia = Melisa
AbueloGervasio.nieta = Melisa

#Si antes de alimentar al nieto preguntáramos Fito.es_feliz_como_su_novia?, respondería false

3.times { AbueloGervasio.alimentar_nieta }
```

... Melisa es un **objeto compartido**: tanto el abuelo como su novio lo conocen. La consecuencia de esto es que cuando su abuelo le da de comer le aumenta la felicidad, y su novio ve los cambios: éste método que antes devolvía `false`, ahora devuelve `true`.

Y esto tiene sentido: si un objeto *muta* su estado, y lo expone de una u otra forma a través de mensajes, todos los que lo observen podrán ver el cambio. ☺

Esta guía fue desarrollada por Felipe Calvo, Franco Bulgarelli bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Para cerrar

Antes de terminar nos topamos con un último problema: Jazmín toca el piano familiar, pero con el uso se va desafinando, y Lucio, el afinador, tiene que afinarlo. En particular:

- Cada vez que Jazmín toca, el nivel de afinación del piano (**initialmente en 100**) baja en un 1%
- El piano está afinado si su nivel de afinación está por encima del 80%
- Cada vez que Lucio afina al piano, su nivel de afinación aumenta tanto como tiempo le dedique: 5% por cada hora. Y nunca se pasa del 100%, claro.

Desarrollá los objetos necesarios para que podamos hacer lo siguiente:

```
#Configura al piano de Jazmin
Jazmin.piano=(PianoFamiliar)
Jazmin.tocar
#Pregunta si está afinado
PianoFamiliar.está_afinado?
#Lo afina durante 3 horas
Lucio.afinar(PianoFamiliar, 3)
```

[Dame una pista!](#)

Obviamente la [afinación de pianos](#) (https://es.wikipedia.org/wiki/Afinaci%C3%B3n_del_piano) es un poco más compleja que lo que dice el enunciado Pero recordá que este es sólo un modelo de la realidad.

Al desarrollar `PianoFamiliar` tenés la libertad de elegir cómo hacer para que se desafine o se afine. Pero recordá que es importante **delegar** adecuadamente!

Y para que la afinación no se pase de 100 te puede servir el mensaje `min` que entienden las colecciones, y devuelve en nuestro caso el menor número:

```
[17, 10].min
=> 10
```

[Solución](#) [Consola](#)

```
1 module PianoFamiliar
2   @afinacion = 100
3
4   def self.afinacion
5     @afinacion
6   end
7   def self.desafinar
8     @afinacion -= 1
9   end
10  def self.está_afinado?
11    @afinacion > 80
12  end
13  def self.afinar(tiempo)
14    @afinacion = [(@afinacion + tiempo * 5), 100].min
15  end
16 end
17
18 module Jazmin
19
20   def self.piano=(instrumento)
```

```
21 @piano = instrumento
22 end
23 def self.tocar
24   @piano.desafinar
25 end
26 end
27
28 module Lucio
29
30   def self.afinar(instrumento, tiempo)
31     instrumento.afinar(tiempo)
32   end
33 end
34
35
36
```



✓ ¡Muy bien! Tu solución pasó todas las pruebas

Esta guía fue desarrollada por Felipe Calvo, Franco Bulgarelli bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)





Zombi caminante

¡Te damos la bienvenida a la invasión zombi!

Vamos a crear al primero de nuestros zombis: Bouba . Bouba no sabe correr, porque es un simple caminante , y cuando le pedimos que grite, responde " ¡agrrrg! ". Además sabe decirnos su salud , que inicialmente es 100, pero puede cambiar.

¿Cuándo cambia? Al recibir_danio! : cuando lo atacan con ciertos puntos de daño, su salud disminuye el doble de esa cantidad.

Manos a la obra: creá el objeto Bouba , que debe entender los mensajes sabe_correr? , gritar , salud y recibir_danio! .

¡Cuidado! La salud de Bouba no puede ser menor que cero.

[Dame una pista!](#)

¡Recordá que las colecciones entienden el mensaje max ! En el caso de una colección de números, devuelve el más alto:

```
 [-5, 7].max  
=> 7
```

Solución Consola

```
1 module Bouba  
2   @salud = 100  
3  
4   def self.sabe_correr?  
5     false  
6   end  
7   def self.gritar  
8     " ¡agrrrg! "  
9   end  
10  def self.salud  
11    @salud  
12  end  
13  def self.recibir_danio!(puntos)  
14    @salud = [(@salud - puntos*2), 0].max  
15  end  
16 end  
17
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¡Bien! La salud de nuestro zombi Bouba disminuye cuando recibe daño. ¡Pero aún no hay nadie que lo pueda atacar! Acompañanos...



Atacando un zombi

Te presentamos a la primera de las sobrevivientes de la invasión, Juliana 🧟. Por ahora su comportamiento es simple: sabe `atacar!` a un zombi con cierta cantidad de puntos de daño. Y al hacerlo, el zombi `recibe daño`.

Además cuenta con un nivel de `energía`, que inicia en `1000`, pero todavía no haremos nada con él. Declará un `getter` para este atributo.

a
e

Veamos si se entiende: creá el objeto `Juliana` que pueda `atacar!` a un zombi haciéndolo `recibir_danio!`, e inicializá su energía en 1000.

☒ [;Dame una pista!](#)

¿Qué dos cosas tiene que saber `Juliana` para poder `atacar!`? ¡A quién y con cuántos puntos de daño!

☒ [Solución](#) ☒ [Consola](#)

```
1 module Juliana
2   @energia = 1000
3
4   def self.energia
5     @energia
6   end
7   def self.atacar!(zombi, puntos)
8     zombi.recibir_danio!(puntos)
9   end
10 end
11
```

☒ Enviar

☒ **¡Muy bien! Tu solución pasó todas las pruebas**

Ahora que `Juliana` sabe `atacar!`, veamos contra quién más se puede enfrentar... 🤖



Otro zombi caminante

¡Bouba no está solo! Resulta que tiene un amigo, Kiki. Podríamos decir que los dos son tal para cual: ¡el comportamiento de ambos es exactamente el mismo! 🤯 Es decir, no `sabe_correr?`, grita "¡agrrrg!", recibe daño de la misma forma...

Creá otro objeto, Kiki, que se comporte de la misma forma que Bouba. ¡Te dejamos a Bouba para que lo uses como inspiración! 🎉

Solución Consola

```
1 module Bouba
2   @salud = 100
3
4   def self.sabe_correr?
5     false
6   end
7   def self.gritar
8     "¡agrrrg!"
9   end
10  def self.salud
11    @salud
12  end
13  def self.recibir_danio!(puntos)
14    @salud = [(@salud - puntos*2), 0].max
15  end
16 end
17
18 module Kiki
19   @salud = 100
20
21   def self.sabe_correr?
22     false
23   end
24   def self.gritar
25     "¡agrrrg!"
26   end
27   def self.salud
28     @salud
29   end
30   def self.recibir_danio!(puntos)
31     @salud = [(@salud - puntos*2), 0].max
32   end
33 end
34
35
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¿Qué pasó acá? Tenemos dos objetos de comportamiento **idéntico**, cuya única diferencia es la *referencia* con la que los conocemos: uno es Bouba, el otro es Kiki. ¡Pero estamos **repetiendo lógica** en el comportamiento de ambos objetos! 🎉



¡¿Vivos?!

¿Acaso Bouba y Kiki pensaron que eran invencibles? Cuando su salud llega a 0, su vida termina... nuevamente. ¡Son zombis, después de todo!

⊕

Desarrollá el método `sin_vida?` que nos dice si la salud de Bouba o Kiki es cero.

⊗ Solución ⊗ Consola

```
1 module Bouba
2   @salud = 100
3
4   def self.sabe_correr?
5     false
6   end
7   def self.gritar
8     "jagrrrg!"
9   end
10  def self.salud
11    @salud
12  end
13  def self.recibir_danio!(puntos)
14    @salud = [(@salud - puntos*2), 0].max
15  end
16  def self.sin_vida?
17    @salud == 0
18  end
19 end
20
21 module Kiki
22   @salud = 100
23
24   def self.sabe_correr?
25     false
26   end
27   def self.gritar
28     "jagrrrg!"
29   end
30   def self.salud
31    @salud
32  end
33  def self.recibir_danio!(puntos)
34    @salud = [(@salud - puntos*2), 0].max
35  end
36  def self.sin_vida?
37    @salud == 0
38  end
39 end
40
41
```

⊗ Enviar

⊗ ¡Muy bien! Tu solución pasó todas las pruebas

Al igual que nos pasó con el resto de los mensajes, `sin_vida?` es exactamente igual para ambos zombis. ¡Otra vez hubo que escribir todo dos veces! 😳

Ahora ya es imposible no verlo: todo lo que se modifique en un zombi también se modifica en el otro. ¿Qué problemas nos trae esto?

- Aunque nos equivoquemos en *una* cosa, el error se repite *dos* veces.
 - Si cambiara la forma en la que, por ejemplo, reciben daño, tendríamos que reescribir `recibir_danio` dos veces.
 - ¿Y si hubiese **diez** zombis en lugar de dos? ¿Y si hubiese **cien**? ¡Cuántas veces habría que copiar y pegar! 😱

Veamos una solución posible...

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)





Clases

Si tenemos más de un objeto que se comporta **exactamente** de la misma forma, lo que podemos hacer es generalizar ese comportamiento declarando una **clase**. Por ejemplo, si tenemos dos celulares con el mismo saldo y ambos tienen las mismas funcionalidades, `realizar_llamada!` y `cargar_saldo!`:

```
module CelularDeMaria
  @saldo = 25

  def self.realizar_llamada!
    @saldo -= 5
  end

  def self.cargar_saldo!(pesos)
    @saldo += pesos
  end
end

module CelularDeLucrecia
  @saldo = 25

  def self.realizar_llamada!
    @saldo -= 5
  end

  def self.cargar_saldo!(pesos)
    @saldo += pesos
  end
end
```

Podemos generalizarlos en una **clase Celular**:

```
class Celular
  def initialize
    @saldo = 25
  end

  def realizar_llamada!
    @saldo -= 5
  end

  def cargar_saldo!(pesos)
    @saldo += pesos
  end
end
```

Veamos si se entiende: como `Bouba` y `Kiki` se comportan exactamente de la misma forma, **generalizalos** creando una clase `Zombi` que entienda los mismos cinco mensajes que ellos. Podés ver el código de ambos zombis en la solapa Biblioteca.

[.¡Dame una pista!](#)

¡Atención! No todo es tan simple: notá que, a diferencia de en los objetos, los métodos en las clases **no se preceden** con la palabra `self`. ¿Y qué significa `initialize`? ¡Paciencia!

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

```
1 class Zombi
2   def initialize
3     @salud = 100
4   end
5
```

```
6 def sabe_correr?
7     false
8 end
9 def gritar
10    "¡agrrrg!"
11 end
12 def salud
13    @salud
14 end
15 def recibir_danio!(puntos)
16    @salud = [(@salud - puntos*2), 0].max
17 end
18 def sin_vida?
19    @salud == 0
20 end
21 end
```

► Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Q Las clases sólo nos sirven para generalizar objetos que tengan el mismo comportamiento: **mismos métodos y mismos atributos**. En nuestro caso, el código de ambos celulares y de ambos zombis es el mismo, por eso pudimos generalizarlo.

Si el código es parecido pero no puede ser generalizado para que sea el mismo, las clases no nos servirán. Al menos por ahora...

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Instancias

Como habrás visto, definir una clase es muy similar a definir un objeto. Tiene métodos, atributos... ¿cuál es su particularidad, entonces? La clase es un objeto que nos sirve como **molde** para crear nuevos objetos. 😊

Momento, ¿cómo es eso? ¿Una clase puede **crear nuevos objetos**?

¡Así es! Aprovechemos la clase `Celular` para **instanciar** los celulares de María y Lucrecia:

```
celular_de_maría = Celular.new  
celular_de_lucrecia = Celular.new
```

`Celular`, al igual que *todas las clases*, entiende el mensaje `new`, que crea una nueva **instancia** de esa clase.

¡Ahora te toca a vos! Definí `bouba` y `kiki` como **instancias** de la clase `Zombi`.

Solución Biblioteca Consola

```
1 bouba = Zombi.new  
2 kiki = Zombi.new
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¿Por qué ahora escribimos `bouba` en lugar de `Bouba`? ¿O por qué `celular_de_maría` en lugar de `CelularDeMaría`?

Hasta ahora estuvimos jugando con **objetos bien conocidos** ([..../exercises/mumukiproject/mumuki-guia-ruby-referencias/7](#)), como `Pepita` o `Fito`. Esos objetos, al igual que las clases, comienzan en mayúscula. Pero `bouba` y `celular_de_maría` son variables: en particular, son referencias que apuntan a **instancias** de `Zombi` y `Celular`.

Y como ya aprendiste anteriormente ([..../exercises/mumukiproject/mumuki-guia-ruby-referencias/2](#)), las variables como `saludo`, `despedida`, o `kiki` comienzan con minúscula. 😊



Al menos tenemos salud

Quizá hayas notado que nuestra clase `Zombi` tiene, al igual que tuvieron los objetos `Bouba` y `Kiki` en su momento, un atributo `@salud`. Seguramente tu `Zombi` se ve similar a este:

```
class Zombi

  def initialize
    @salud = 100
  end

  def salud
    @salud
  end

  #...y otros métodos

end
```

Pero ahora que `@salud` aparece en la clase `Zombi`, ¿eso significa que comparten el atributo? Si `Juliana` ataca a `bouba`, ¿disminuirá también la salud de `kiki`?

¡Averigualo! Hacé que `Juliana` ataque a cada zombi con distintos puntos de daño y luego consultá la salud de ambos.

¡Dame una pista!

¡Recordá que el mensaje `atacar!` recibe dos parámetros: un zombie y una cantidad de puntos de daño!

Consola Biblioteca

```
Juliana.atacar!(bouba, 25)
=> 50
Juliana.atacar!(kiki, 10)
=> 80
```



Inicializando instancias

Como viste recién, la salud no se comparte entre bouba y kiki a pesar de que ambos sean instancias de Zombi.

Pero nos quedó un método misterioso por aclarar: initialize. Al trabajar con clases tenemos que *inicializar* los atributos en algún lugar. ¡Para eso es que existe ese método!

El mensaje initialize nos permite especificar **cómo queremos que se inicialice** la instancia de una clase. ¡Es así de fácil! 🎉

¡anastasia llega para combatir los zombis! Declará una clase Sobreviviente que sepa atacar! zombis e inicialice la energía en 1000. En la solapa Biblioteca podés ver el código de la Juliana original.

Luego, definí juliana y anastasia como instancias de la nueva clase Sobreviviente.

Solución Biblioteca Consola

```
1 class Sobreviviente
2   def initialize
3     @energia = 1000
4   end
5   def energia
6     @energia
7   end
8   def atacar!(zombi, puntos)
9     zombi.recibir_danio!(puntos)
10  end
11 end
12
13 juliana = Sobreviviente.new
14 anastasia = Sobreviviente.new
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



Ahora sí: invasión

Prometimos una invasión zombi pero sólo tenemos dos . Ahora que contamos con un molde para crearlos fácilmente, la clase `Zombi`, podemos hacer zombis *de a montones*.

¿Eso significa que tenés que pensar un nombre para referenciar a cada uno? ¡No! Si, por ejemplo, agregamos algunas plantas a un `Vivero` ...

```
Vivero.agregar_planta!(Planta.new)
Vivero.agregar_planta!(Planta.new)
Vivero.agregar_planta!(Planta.new)
```

...y el `Vivero` las guarda en una colección `@plantas`, luego las podemos regar a todas...

```
def regar_todas!
  @plantas.each { |planta| planta.regar! }
end
```

...a pesar de que no tengamos una *referencia explícita* para cada planta. ¡Puede ocurrir que no necesitemos darle un nombre a cada una!

Veamos si se entiende: Agregale veinte nuevos zombis a la colección `caminantes`. ¡No olvides que los números entienden el mensaje `times`!

Luego, agregale un método `ataque_masivo!` a `Sobreviviente`, que reciba una colección de zombis y los ataque a todos con 15 puntos de daño.

¡Dame una pista!

Para agregar zombis a la colección `caminantes` recordá que podés enviarle el mensaje `push` a la colección.

Solución Biblioteca Consola

```
1 class Sobreviviente
2   def initialize
3     @energia = 1000
4   end
5   def energia
6     @energia
7   end
8   def atacar!(zombi, puntos)
9     zombi.recibir_danio!(puntos)
10  end
11  def ataque_masivo!(zombis)
12    zombis.each { |zombi| atacar!(zombi, 15)}
13  end
14 end
15
16 juliana = Sobreviviente.new
17 anastasia = Sobreviviente.new
18
19 class Zombi
20   def initialize
21     @salud = 100
22   end
23
24   def sabe_correr?
25     false
26   end
27   def gritar
28     "¡agrrrg!"
```

```
29 end
30 def salud
31   @salud
32 end
33 def recibir_danio!(puntos)
34   @salud = [(@salud - puntos*2), 0].max
35 end
36 def sin_vida?
37   @salud == 0
38 end
39 end
40
41 caminantes = []
42 20.times { caminantes.push(Zombi.new) }
43
```

► Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

¡De acuerdo! Es importante tener en cuenta que nuestros objetos **también pueden crear otros objetos**, enviando el mensaje `new` a la clase que corresponda.

Por lo tanto, los casos en los que un objeto puede conocer a otro son:

- Cuando es un **objeto bien conocido**, como con los que veníamos trabajando hasta ahora
- Cuando el objeto se pasa por parámetro en un mensaje (`Juliana.atacar(bouba, 4)`)
- Cuando un objeto crea otro mediante el envío del mensaje `new`

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  Mumuki (<http://mumuki.org/>)





Al menos tenemos (menos) salud

juliana y anastasia estuvieron estudiando a los zombis y descubrieron que no todos gozan de máxima vitalidad: algunos de ellos tienen menos salud que lo que pensábamos. ☺

¡Esto es un gran inconveniente! En nuestra clase `Zombi`, todos se inicializan con `@salud = 100`. ¿Cómo podemos hacer si necesitamos que alguno de ellos inicie con 90 de `@salud`? ¿Y si hay otro con 80? ¿Y si hay otro con 70? No vamos a escribir una clase nueva para cada caso, ¡estaríamos repitiendo toda la lógica de su comportamiento! ☹

Afortunadamente el viejo y querido `initialize` puede recibir parámetros que especifiquen **con qué valores** deseamos inicializar los atributos al construir nuestros objetos. ¡Suena ideal para nuestro problema!

```
class Planta
  @altura

  def initialize(centimetros)
    @altura = centimetros
  end

  def regar!
    @altura += 2
  end
end
```

Ahora podemos crear plantas cuyas alturas varíen utilizando una única clase. Internamente, los parámetros que recibe `new` se pasan también a `initialize`:

```
brote = Planta.new(2)
arbusto = Planta.new(45)
arbolito = Planta.new(110)
```

¡Y de esa forma creamos tres plantas de 2 ☺, 45 ☺ y 110 ☺ centímetros de `@altura`!

¡Ahora te toca a vos! Modificá la clase `Zombi` para que `initialize` pueda recibir la salud inicial del mismo.

Solución Consola

```
1 class Zombi
2   @salud
3
4   def initialize(nivel_de_salud)
5     @salud = nivel_de_salud
6   end
7
8   def sabe_correr?
9     false
10  end
11  def gritar
12    "jagrrrg!"
13  end
14  def salud
15    @salud
16  end
17  def recibir_danio!(puntos)
18    @salud = [(@salud - puntos*2), 0].max
19  end
20  def sin_vida?
21    @salud == 0
```

► Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Lo que hiciste recién en la clase `Zombi` fue **especificar un constructor**: decirle a la clase cómo querés que se construyan sus instancias.

Los constructores pueden recibir más de un parámetro. Por ejemplo, si de una `Planta` no sólo pudiéramos especificar su altura, sino también su especie y si da o no frutos...

```
jazmin = Planta.new(70, "Jasminum fruticans", true)
```



Esta guía fue desarrollada por Felipe Calvo bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)



(<https://www.facebook.com/felipecalvo11>)



Súper zombi

Finalmente llegó el momento que más temíamos: ¡algunos zombis aprendieron a correr y hasta a recuperar salud! Y esto no es un problema para las sobrevivientes únicamente, sino para nosotros también. Ocurre que los súper zombis saben hacer las mismas cosas que los comunes, pero las hacen de forma distinta. ¡No nos alcanza con una única clase `Zombi`! 😱

Un `SuperZombi` `sabe_correr?` 🏃, y en lugar del doble, recibe el **triple de puntos de daño**. Sin embargo, puede gritar y decirnos su salud de la misma forma que un `Zombi` común, y queda `sin_vida?` en los mismos casos: cuando su `salud` es 0.

Pero eso no es todo, porque también pueden regenerarse!. Al hacerlo, su `salud` vuelve a 100.

¡A correr! Definí la clase `SuperZombi` aplicando las modificaciones necesarias a la clase `Zombi`.

Solución Biblioteca Consola

```

1 class SuperZombi
2
3   def initialize(nivel_de_salud)
4     @salud = nivel_de_salud
5   end
6   def sabe_correr?
7     true
8   end
9   def gritar
10    "¡agrrrg!"
11  end
12  def salud
13    @salud
14  end
15  def recibir_danio!(puntos)
16    @salud = [(@salud - puntos*3), 0].max
17  end
18  def sin_vida?
19    @salud == 0
20  end
21  def regenerarse!
22    @salud = 100
23  end
24 end
25

```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Veamos por qué decidimos hacer una nueva clase, `SuperZombi`:

- Pueden regenerarse!, a diferencia de un `Zombi`
- `sabe_correr?` tiene comportamiento distinto a la clase `Zombi`
- `recibir_danio!` tiene comportamiento distinto a la clase `Zombi`

Sin embargo habrás notado que, aunque esos últimos dos métodos son distintos, hay **cuatro** que son idénticos: `salud`, `gritar`, `sin_vida?`, y su inicialización mediante `initialize`. ¡Hasta tienen un mismo atributo, `@salud`! ¿Acaso eso no significa que estamos repitiendo mucha lógica en ambas clases? 😬

¡Así es! Pero todavía no contamos con las herramientas necesarias para solucionarlo. ☺

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  Mumuki (<http://mumuki.org/>)





Ejercitando

¡Defenderse de la invasión no es para cualquiera! Las sobrevivientes descubrieron que cada vez que realizan un `ataque_masivo!` su energía disminuye a la mitad.

Pero también pueden beber! beber bebidas energéticas para recuperar las fuerzas: cada vez que beben, su `energia` aumenta un 25%.

Modificá la clase `Sobreviviente` para que pueda disminuirse y recuperarse su `energia`.

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

```
1 class Sobreviviente
2   def initialize
3     @energia = 1000
4   end
5
6   def energia
7     @energia
8   end
9
10  def atacar!(zombie, danio)
11    zombie.recibir_danio!(danio)
12  end
13
14  def ataque_masivo!(zombis)
15    zombis.each { |zombi| atacar!(zombi, 15) }
16    @energia *= 0.5
17  end
18  def beber!
19    @energia *= 1.25
20  end
21 end
```

[Enviar](#)

¡Muy bien! Tu solución pasó todas las pruebas

¡Ya casi terminamos! 🎉 Antes de irnos, veamos un tipo de sobreviviente distinto...



Aliados

¡Nadie lo esperaba, pero igualmente llegó! 😊 Un Aliado se comporta parecido a una Sobreviviente, pero su `ataque_masivo!` es más violento: brinda 20 puntos de daño en lugar de 15.

Por otro lado, su `energia` inicial es de solamente 500 puntos, y disminuye un 5% al `atacar!`. Y además, `beber!` les provee menos energía: solo aumenta un 10%.

Nuevamente, Sobreviviente y Aliado tienen comportamiento similar **pero no idéntico**: no podemos unificarlo en una única clase. ¡Incluso hay porciones de lógica que se repiten y otras que no en un mismo método! Por ejemplo, en `ataque_masivo!`, los puntos de daño varían, pero el agotamiento es el mismo para ambas clases.

Definí la clase `Aliado`. Podés ver a `Sobreviviente` en la solapa Biblioteca.

Solución Biblioteca Consola

```
1 class Aliado
2   def initialize
3     @energia = 500
4   end
5   def energia
6     @energia
7   end
8   def atacar!(zombi, puntos)
9     zombi.recibir_danio!(puntos)
10    @energia *= 0.95
11  end
12  def ataque_masivo!(zombis)
13    zombis.each { |zombi| atacar!(zombi, 20)}
14    @energia *= 0.5
15  end
16  def beber!
17    @energia *= 1.1
18  end
19 end
20
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Solución

Biblioteca

Consola

```
class Sobreviviente
  def initialize
    @energia = 1000
  end

  def energia
    @energia
  end

  def atacar!(zombie, danio)
    zombie.recibir_danio!(danio)
  end

  def ataque_masivo!(zombis)
    zombis.each { |zombi| atacar!(zombi, 15) }
    @energia *= 0.5
  end
  def beber!
    @energia *= 1.25
  end
end
```



Auto

¡Es la hora de movilizarse! Todos los días la gente necesita trasladarse a distintos lugares, ya sea a pie, en tren, en bicicleta... Nosotros no podíamos quedarnos atrás: vamos a implementar nuestros propios medios de transporte (https://es.wikipedia.org/wiki/Anexo:Medios_de_transporte) en Objetos.

Empecemos por uno simple, el Auto 🚗. Las instancias de la clase Auto se inicializan con 40 @litros de combustible.

¿Qué sabe hacer un Auto? Puede decirnos si es ligero?, que es verdadero cuando la cantidad de @litros es menor a 20. Su combustible disminuye a medida que se desplaza: se puede conducir! una cierta cantidad de kilómetros, y los @litros disminuyen a razón de 0.05 por cada kilómetro.

También sabe responder su cantidad_de_ruedas : siempre es 4, porque no contamos la rueda de auxilio 😊 .

Veamos si se entiende: definí la clase Auto, que sepa entender los mensajes initialize, ligero?, conducir! y cantidad_de_ruedas .

☒ ¡Dame una pista!

No nos vamos a preocupar aún en que haya suficientes @litros de combustible para poder conducir una cantidad de kilómetros. ¡Más adelante vamos a tener herramientas para evitar que alguien conduzca con el tanque vacío! 😅

☒ Solución ☒ Consola

```
1 class Auto
2
3     def initialize
4         @litros = 40
5     end
6     def ligero?
7         @litros < 20
8     end
9     def conducir!(kilometros)
10        @litros -= kilometros*0.05
11    end
12    def cantidad_de_ruedas
13        4
14    end
15 end
16
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas

☒ Vamos de paseo, en un auto feo... ☒

¡Ya tenemos el primero de nuestros medios de transporte! Ahora, pasemos a algo con la mitad de ruedas...



Moto

¡Ahora es el turno de la Moto! 🚲

La clase Moto entiende los mismos mensajes que Auto, pero no se comporta igual. Por ejemplo, se inicializa con 20 @litros, y es ligero? cuando tiene menos de 10.

Consumo más combustible: 0.1 por cada kilómetro al conducir! una distancia en kilómetros. Y, lógicamente, su cantidad_de_ruedas es 2.

Pará la moto: definí la clase Moto, que sepa entender los mensajes initialize, ligero?, conducir! y cantidad_de_ruedas.

Solución Biblioteca Consola

```
1 class Moto
2
3   def initialize
4     @litros = 20
5   end
6   def ligero?
7     @litros < 10
8   end
9   def conducir!(kilometros)
10    @litros -= kilometros*0.1
11  end
12  def cantidad_de_ruedas
13    2
14  end
15 end
16
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¿Acaso para la Moto no deberíamos preguntar si es ligera? en lugar de ligero? ?

¡Puede ser! Pero si los mensajes se llaman distinto, no podemos tratar polimórficamente a los objetos. Por ejemplo, no podríamos saber cuántos medios de transporte ligeros hay en una colección de autos y motos, porque no habría un único mensaje -tendríamos ligero? y ligera? - que respondiera nuestra pregunta.

```
☒ transportes.count { |transporte| transporte.ligero? }
=> #¡Falla porque Moto no entiende el mensaje ligero?!

☒ transportes.count { |transporte| transporte.ligera? }
=> #¡Falla porque Auto no entiende el mensaje ligera?!
```



Clase abstracta

Q Una forma de organizar las clases cuando programamos en Objetos es definir una **jerarquía**. En nuestro caso podemos pensar que Moto y Auto se pueden englobar en algo más grande y que las incluye, la idea de MedioDeTransporte .

Muchas veces esa jerarquía se puede visualizar en el mundo real: por ejemplo, Perro y Gato entran en la categoría Mascota , mientras que Cónedor y Halcón se pueden clasificar como Ave . Cuando programemos, la jerarquía que utilicemos dependerá de nuestro modelo y de las abstracciones que utilicemos.

Si tenemos abstracciones para Moto y Auto , ¿alguna vez instanciaremos un objeto de la clase MedioDeTransporte ? ¡Probablemente no! ¿Por qué queríamos ser tan genéricos con nuestras clases si podemos ser específicos?

En el ejemplo con animales ocurre parecido: si definimos implementaciones específicas para Cónedor , Halcón , Perro y Gato , no va a haber un objeto de la clase Ave o Mascota en nuestro sistema.

A esas clases, como MedioDeTransporte o Ave , se las llama **clases abstractas** porque, a diferencia de las **clases concretas** (como Moto o Auto), nunca las instanciamos, en criollo, no creamos objetos con esa clase. Sirven para especificar qué métodos deben implementar aquellas clases que estén más *abajo* en la jerarquía.

```
class Ave
  def volar!
  end
end

class Condor < Ave
  def volar!
    @energia -= 20
  end
end

class Halcon < Ave
  def volar!
    @energia -= 35
  end
end
```

El símbolo < significa "hereda de": por ejemplo, Cónedor hereda de Ave , que está *más arriba* en la jerarquía. En la clase abstracta Ave , el método volar! **no tiene comportamiento** porque el comportamiento lo implementan las clases concretas Halcón y Cónedor . Entonces, decimos que volar! es un **método abstracto**.

¡Uf! ¡Eso fue un montón! 😅 A ver si quedó claro. Marcá las opciones correctas:

- MedioDeTransporte es una clase abstracta.
- Mascota es una clase concreta.
- Las clases abstractas no se instancian.
- Mascota hereda de Gato .
- Moto hereda de MedioDeTransporte .
- Un método abstracto no tiene comportamiento.

Enviar

¡La respuesta es correcta!

This document is available free of charge on



Medio de transporte

Ahora que ya conocimos a las clases abstractas, ¡pongamos manos a la obra! 🤲

Definí la clase abstracta `MedioDeTransporte` y sus métodos abstractos `ligero?`, `conducir!` y `cantidad_de_ruedas`.

Luego, hacé que las clases `Moto` y `Auto` hereden de `MedioDeTransporte`.

☒ [Dame una pista!](#)

Ignoramos el método `initialize` porque ya lo entienden todas las clases: no es necesario especificar que `Moto` y `Auto` lo implementen.

☒ [Solución](#) ☒ [Consola](#)

```
1 class MedioDeTransporte
2   def ligero?
3   end
4   def conducir!(kilometros)
5   end
6   def cantidad_de_ruedas
7   end
8 end
9
10 class Auto < MedioDeTransporte
11
12   def initialize
13     @litros = 40
14   end
15   def ligero?
16     @litros < 20
17   end
18   def conducir!(kilometros)
19     @litros -= kilometros*0.05
20   end
21   def cantidad_de_ruedas
22     4
23   end
24 end
25
26
27 class Moto < MedioDeTransporte
28
29   def initialize
30     @litros = 20
31   end
32   def ligero?
33     @litros < 10
34   end
35   def conducir!(kilometros)
36     @litros -= kilometros*0.1
37   end
38   def cantidad_de_ruedas
39     2
40   end
41 end
42
```

☒ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Ahora que nuestra jerarquía de transportes empieza a tomar forma, podemos darle un nombre a dos ideas que surgieron.

En primer lugar, `MedioDeTransporte` es la **superclase** de `Auto` y `Moto`, porque está *más arriba* en la jerarquía.

Y a su vez, `Moto` y `Auto` son **subclases** de `MedioDeTransporte`, porque heredan de ella. ¡Así de simple! 😊

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Todo pesa

;Se sumó una funcionalidad que no habíamos tenido en cuenta! 🤔 Necesitamos que Auto y Moto nos sepan decir su peso en kilogramos. Afortunadamente, la cuenta es simple: el peso es igual a la cantidad_de_ruedas multiplicado por 200.

A ver si quedó claro: Hacé que Auto y Moto entiendan el mensaje peso . ;No modifiques la clase MedioDeTransporte !

[Solución](#) [Consola](#)

```
1 class MedioDeTransporte
2   def ligero?
3   end
4   def conducir!(kilometros)
5   end
6   def cantidad_de_ruedas
7   end
8 end
9
10 class Auto < MedioDeTransporte
11
12   def initialize
13     @litros = 40
14   end
15   def ligero?
16     @litros < 20
17   end
18   def conducir!(kilometros)
19     @litros -= kilometros*0.05
20   end
21   def cantidad_de_ruedas
22     4
23   end
24   def peso
25     @peso = cantidad_de_ruedas * 200
26   end
27 end
28
29
30 class Moto < MedioDeTransporte
31
32   def initialize
33     @litros = 20
34   end
35   def ligero?
36     @litros < 10
37   end
38   def conducir!(kilometros)
39     @litros -= kilometros*0.1
40   end
41   def cantidad_de_ruedas
42     2
43   end
44   def peso
45     @peso = cantidad_de_ruedas * 200
46   end
47 end
48
```

[Enviar](#)

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Todo muy lindo, tenemos cosas que pesan, pero... ¡estamos repitiendo lógica! El cálculo de peso está dos veces, una en Auto y otra en Moto.
¡Solucionemos eso!

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Peso repetido

Una ventaja de la herencia es que nos permite agrupar comportamiento en la superclase que, de otra forma, tendríamos repetido en las subclases. ¡Es exactamente lo que nos ocurre con `peso`! ¡Qué casualidad! 😊

Como el cálculo de `peso` es igual para todos los `MedioDeTransporte` que tenemos, podemos **pasarlo tal como está a la superclase** y, como `Auto` y `Moto` heredan de ella, van a seguir entendiendo el mensaje de la misma forma que antes. ¡En serio!

Nada de lo anterior cambia: `MedioDeTransporte` sigue siendo una clase abstracta, pero en ella, `peso` es un **método concreto**, ya que se define su comportamiento para todas las subclases. ¡Y dejamos de repetir esa lógica!

Veamos si se entiende: quítá el método `peso` de `Auto` y `Moto` y agregalo a `MedioDeTransporte`.

☒ Solución ☒ Consola

```
1 class MedioDeTransporte
2   def ligero?
3   end
4   def conducir!(kilometros)
5   end
6   def cantidad_de_ruedas
7   end
8   def peso
9     @peso = cantidad_de_ruedas * 200
10  end
11 end
12
13 class Auto < MedioDeTransporte
14
15   def initialize
16     @litros = 40
17   end
18   def ligero?
19     @litros < 20
20   end
21   def conducir!(kilometros)
22     @litros -= kilometros*0.05
23   end
24   def cantidad_de_ruedas
25     4
26   end
27 end
28
29
30 class Moto < MedioDeTransporte
31
32   def initialize
33     @litros = 20
34   end
35   def ligero?
36     @litros < 10
37   end
38   def conducir!(kilometros)
39     @litros -= kilometros*0.1
40   end
41   def cantidad_de_ruedas
42     2
43   end
44 end
```

► Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

¿No es *casi magia*? A pesar de que Moto y Auto no tienen definido un método peso , siguen entendiendo el mensaje porque lo heredan de su superclase MedioDeTransporte . ¡Fantástico! 😊

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  Mumuki (<http://mumuki.org/>)





Bicicleta

Silenciosamente se acerca el transporte más ecológico de todos: la `Bicicleta`. 🚴 ¿Qué sabe hacer una `Bicicleta`? ¡Lo mismo que cualquier `MedioDeTransporte`, obvio! Por eso hereda de esa clase. Pero su comportamiento ya no es tan similar al de `Auto` y `Moto`.

Para empezar, su `cantidad_de_ruedas` es 2, y siempre es `ligero?`. No lleva combustible, por lo que ya no lleva cuenta de los litros: al `conducir!` una cantidad de kilómetros, los suma a una cuenta de `@kilometros_recorridos`. Al inicializarse, una `Bicicleta` lleva 0 `@kilometros_recorridos`.

Todo va sobre ruedas: definí una clase `Bicicleta` que herede de `MedioDeTransporte` y que entienda los mensajes `initialize`, `cantidad_de_ruedas`, `ligero?` y `conducir!`.

Solución Biblioteca Consola

```
1 class Bicicleta < MedioDeTransporte
2   def initialize
3     @kilometros_recorridos = 0
4   end
5   def ligero?
6     true
7   end
8   def kilometros_recorridos
9     @kilometros_recorridos
10  end
11  def cantidad_de_ruedas
12    2
13  end
14  def conducir!(kilometros)
15    @kilometros_recorridos += kilometros
16  end
17 end
18
19
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



Muy pesada

Antes de continuar, comprobemos lo que hablamos antes. Si bien el código de peso no está en las clases concretas Auto, Moto y Bicicleta, las tres entienden el mensaje porque heredan de una clase que lo entiende, MedioDeTransporte .

Para este caso creamos los objetos fitito, mel y playera, que son instancias de Auto, Moto y Bicicleta respectivamente, para que puedas enviarles el mensaje peso .

¡Compruébalo! ¿Hay algún peso que no tenga mucho sentido? 😊

```
fitito.peso  
=> 800  
mel.peso  
=> 400  
playera.peso  
=> 400  
 
```

☒



Redefiniendo peso

¡¿400 kilos una Bicicleta?! ¡¿Cómo hacés para trasladarla?! 🤔

Algo en nuestro modelo falló: como la Bicicleta tiene dos ruedas, y es un MedioDeTransporte, su peso se calcula multiplicado 200 por 2. ¡Pero no puede pesar tanto!

Sin embargo, no queremos que deje de heredar de MedioDeTransporte. Lo que podemos hacer es **redefinir** el método: si Bicicleta implementa el método peso, es ese el que se va a evaluar en lugar del de su superclase.

En lugar de que MedioDeTransporte realice el cálculo, le agregamos a la propia Bicicleta el método peso, que lo calculará como la cantidad de ruedas multiplicado por 3.

Veamos si se entiende: hacé que Bicicleta implemente el método peso.

Solución Consola

```
1 class Bicicleta < MedioDeTransporte
2   def initialize
3     @kilometros_recorridos = 0
4   end
5   def ligero?
6     true
7   end
8   def kilometros_recorridos
9     @kilometros_recorridos
10  end
11  def cantidad_de_ruedas
12    2
13  end
14  def conducir!(kilometros)
15    @kilometros_recorridos += kilometros
16  end
17  def peso
18    @peso = cantidad_de_ruedas * 3
19  end
20 end
21
22
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¡Genial! 🎉

Redefinir el método de una superclase, como hicimos con peso, nos permite **modificar el comportamiento** que definió la superclase originalmente. De ese modo, declarar peso en MedioDeTransporte nos permite agrupar la lógica unificada de Auto y Moto - pero a la vez, podemos contemplar los casos en los que requerimos otro comportamiento, como en Bicicleta.



El regreso de los zombis

¿Creíste que habíamos terminado con los zombis? ¡Nada más alejado de la realidad! 😱

Cuando surgieron los SuperZombi (<https://staging.mumuki.io/exercises/4297-programacion-con-objetos-clases-e-instancias-super-zombi>), notamos que parte de su comportamiento era compartido con un Zombi común: ambos pueden gritar, decirnos su salud, y responder si están sin_vida? de la misma forma. Pero hasta allí llegan las similitudes: recibir_danio! y sabe_correr? tienen implementaciones distintas, y además, un SuperZombi puede regenerarse!, a diferencia de un Zombi.

¡Esto nos da una nueva posibilidad! Podemos hacer que SuperZombi herede de Zombi para:

- Evitar **repetir la lógica** de aquellos métodos que son iguales, ya que se pueden implementar únicamente en la superclase Zombi
- **Redefinir** en SuperZombi aquellos métodos cuya implementación sea distinta a la de Zombi
- Implementar **únicamente** en SuperZombi el comportamiento que es exclusivo a esa clase

¿Te animás? ¡Marcá las respuestas correctas!

- Zombi debe implementar el método gritar
- Zombi debe implementar el método salud
- Zombi debe implementar el método sin_vida?
- Zombi debe implementar el método recibir_danio!
- Zombi debe implementar el método sabe_correr?
- Zombi debe implementar el método regenerarse!
- SuperZombi debe implementar el método gritar
- SuperZombi debe implementar el método salud
- SuperZombi debe implementar el método sin_vida?
- SuperZombi debe implementar el método recibir_danio!
- SuperZombi debe implementar el método sabe_correr?
- SuperZombi debe implementar el método regenerarse!

✉ Enviar

✉ ¡La respuesta es correcta!



Herencia zombie

¡Todo listo! 🎉 Ahora que sabés qué métodos van en qué clases, es momento de implementar los cambios.

Veamos si se entiende: hacé que la clase `SuperZombi` herede de `Zombi` y modificala para que implemente únicamente los métodos cuyo comportamiento varía respecto de `Zombi`. ¡Notá que la inicialización también es igual en ambas clases!

Solución Consola

```

1 class Zombi
2   def initialize(salud_inicial)
3     @salud = salud_inicial
4   end
5
6   def salud
7     @salud
8   end
9
10  def gritar
11    "jagrrrg!"
12  end
13
14  def sabe_correr?
15    false
16  end
17
18  def sin_vida?
19    @salud == 0
20  end
21
22  def recibir_danio!(puntos)
23    @salud = [@salud - puntos * 2, 0].max
24  end
25 end
26
27 class SuperZombi < Zombi
28
29   def sabe_correr?
30     true
31   end
32
33   def recibir_danio!(puntos)
34     @salud = [@salud - puntos * 3, 0].max
35   end
36
37   def regenerarse!
38     @salud = 100
39   end
40 end

```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

⚠ Prestá atención: lo que hicimos aquí es *parecido* a la herencia de los transportes, pero no igual. En nuestro ejemplo anterior, `MedioDeTransporte` es una clase abstracta, porque nunca la vamos a instanciar, y nuestros tres transportes heredan de ella.

Aquí, sin embargo, `Zombi` no es abstracta sino concreta - y `SuperZombi` hereda de ella sin problemas. ¡Esto significa que en nuestro sistema podemos tener tanto objetos `SuperZombi` como `Zombi`! En este caso, y al igual que con los transportes, las instancias de `SuperZombi` entenderán todos los mensajes que estén definidos en su clase, sumados a todos los que defina `Zombi`.

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)





Micro

¡Estamos de vuelta con un nuevo `MedioDeTransporte`! `Micro` 🚙 también es una de sus subclases, y tiene algunas variaciones. En principio, además de inicializarse con 100 `@litros`, arranca con cero `@pasajeros`.

Al `conducir!` una cierta distancia gasta 0.2 `@litros` por cada kilómetro, y es `ligero?` cuando no lleva `@pasajeros`. Su `cantidad_de_ruedas` es 6.

Veamos si se entiende: definí la clase `Micro`, que hereda de `MedioDeTransporte`, y entiende los mensajes `initialize`, `conducir!`, `ligero?` y `cantidad_de_ruedas`.

Solución Biblioteca Consola

```
1 class Micro < MedioDeTransporte
2   def initialize
3     @litros = 100
4     @pasajeros = 0
5   end
6   def conducir!(kilometros)
7     @litros -= kilometros * 0.2
8   end
9   def cantidad_de_ruedas
10    6
11  end
12  def ligero?
13    @pasajeros == 0
14  end
15 end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

`Micro`, como cualquier otra subclase de `MedioDeTransporte`, también entiende el mensaje `peso` porque de ella lo hereda.

```
 micro_larga_distancia = Micro.new
 micro_larga_distancia.peso
=> 1200
```

Sin embargo, 1200 kilos es poco peso para un micro. ¡En un par de ejercicios volveremos a eso! 😊



Suben y bajan

¿Y cuál es la gracia de tener un Micro si no puede trasladar a nadie? Micro debe entender también los mensajes `sube_pasajero!` y `baja_pasajero!`, que incrementan o disminuyen en uno la cantidad de @pasajeros a bordo.

¡Momento! 📺 ¿Por qué no definirlo en la clase abstracta `MedioDeTransporte`? Porque muchas clases heredan de ella, y **no nos interesa** que el resto de los medios de transporte implementen la lógica de traslado de pasajeros. ¡No entran más de dos personas en una Moto o Bicicleta!

Es tu turno: agregá los métodos `sube_pasajero!` y `baja_pasajero!` a la clase `Micro`.

Solución Consola

```
1 class Micro < MedioDeTransporte
2   def initialize
3     @litros = 100
4     @pasajeros = 0
5   end
6   def conducir!(kilometros)
7     @litros -= kilometros * 0.2
8   end
9   def cantidad_de_ruedas
10    6
11  end
12  def ligero?
13    @pasajeros == 0
14  end
15  def sube_pasajero!
16    @pasajeros += 1
17  end
18  def baja_pasajero!
19    @pasajeros -= 1
20  end
21 end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

A diferencia del resto de los transportes, Micro entiende dos mensajes que Auto, Moto y Bicicleta no: `sube_pasajero!` y `baja_pasajero!` son exclusivos a esa clase.

¡Sigue en pie la idea de que en la superclase `MedioDeTransporte` va únicamente el comportamiento que es común a todas sus subclases!



La pesada herencia

¡Nuevamente tenemos problemas con el `peso`! 😱 No alcanza con que el `Micro` lo calcule utilizando únicamente sus ruedas, porque descubrimos que además depende de la cantidad de `@pasajeros` que esté trasladando.

Y eso nos pone en un problema interesante: de la forma actual, el peso está mal calculado. Pero redefinir `peso` en `Micro` implicaría repetir la lógica de `cantidad_de_ruedas * 200`. ¿Hay otra posibilidad?

¡Sí! El mensaje `super`. Al utilizar `super` en el método de una subclase, se evalúa el método con el mismo nombre de su superclase. Por ejemplo...

```
class Saludo
  def saludar
    "Buen día"
  end
end

class SaludoFormal < Saludo
  def saludar
    super + " señoras y señores"
  end
end
```

De esta forma, al enviar el mensaje `saludar` a `SaludoFormal`, `super` invoca el método `saludar` de su superclase, `Saludo`. 🎉

```
mi_saludo = SaludoFormal.new
mi_saludo.saludar
=> "Buen día señoras y señores"
```

¡Ahora te toca a vos! Agregá el método `peso` a `Micro`, de modo que se calcule como la `cantidad_de_ruedas` multiplicado por 200, sumado a la cantidad de `@pasajeros` por 80. ¡Recordá utilizar `super` para evitar repetir lógica!

☒ Solución ☒ Biblioteca ☒ Consola

```
1 class Micro < MedioDeTransporte
2   def initialize
3     @litros = 100
4     @pasajeros = 0
5   end
6   def conducir!(kilometros)
7     @litros -= kilometros * 0.2
8   end
9   def cantidad_de_ruedas
10    6
11  end
12  def ligero?
13    @pasajeros == 0
14  end
15  def sube_pasajero!
16    @pasajeros += 1
17  end
18  def baja_pasajero!
19    @pasajeros -= 1
20  end
21  def peso
22    cantidad_de_ruedas * 200
23    super + @pasajeros * 80
24  end
25 end
```

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Utilizar `super` nos permite redefinir un método pero sólo agregar **una parte** de nueva funcionalidad, reutilizando la lógica común que está definida en la superclase.

¡Ya casi terminamos! 🎉

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)



```
class MedioDeTransporte
  def ligero?
  end
  def conducir!(kilometros)
  end
  def cantidad_de_ruedas
  end
  def peso
    @peso = cantidad_de_ruedas * 200
  end
end

class Auto < MedioDeTransporte

  def initialize
    @litros = 40
  end
  def ligero?
    @litros < 20
  end
  def conducir!(kilometros)
    @litros -= kilometros*0.05
  end
  def cantidad_de_ruedas
    4
  end
end

class Moto < MedioDeTransporte

  def initialize
    @litros = 20
  end
  def ligero?
    @litros < 10
  end
  def conducir!(kilometros)
    @litros -= kilometros*0.1
  end
  def cantidad_de_ruedas
    2
  end
end
```



Súper ataque

¡Suficientes ruedas por hoy! Para terminar, volvamos un momento a la invasión zombi. Veamos parte del comportamiento de `Sobreviviente` y `Aliado`:

```
class Sobreviviente
  def initialize
    @energia = 1000
  end

  def energia
    @energia
  end

  def beber!
    @energia *= 1.25
  end

  def atacar!(zombi, danio)
    zombi.recibir_danio!(danio)
  end
end

class Aliado
  def initialize
    @energia = 500
  end

  def energia
    @energia
  end

  def beber!
    @energia *= 1.10
  end

  def atacar!(zombi, danio)
    zombi.recibir_danio!(danio)
    @energia *= 0.95
  end
end
```

Como verás, tenemos distintos grados de similitud en el código:

- `energia` es igual para ambas clases, porque sólo devuelve la energía;
- Una parte de `atacar!` coincide: en la que el zombi `recibe_danio!`, pero en `Aliado` reduce energía y en `Sobreviviente` no;
- `beber!` es diferente para ambas clases.

Último esfuerzo: definí una clase abstracta `Persona` que agrupe el comportamiento que se repite y hacé que las clases `Sobreviviente` y `Aliado` hereden de ella. Finalmente, implementá en esas clases el código que es propio de cada una de ellas. ¡No olvides usar `super`!

[.Dame una pista!](#)

Recordá que, a pesar de que las subclases realizan la acción `beber!` de forma distinta, queremos explicitar en la clase abstracta `Persona` que deben implementar ese método.

[Solución](#) [Consola](#)

```
1 class Persona
2   def energia
3     @energia
4   end
```

```
5 def beber!
6 end
7 def atacar!(zombi, danio)
8 end
9 end
10
11 class Sobreviviente < Persona
12   def initialize
13     @energia = 1000
14   end
15   def beber!
16     @energia *= 1.25
17   end
18   def atacar!(zombi, danio)
19     zombi.recibir_danio!(danio)
20   end
21 end
22
23 class Aliado < Persona
24   def initialize
25     @energia = 500
26   end
27   def beber!
28     @energia *= 1.10
29   end
30   def atacar!(zombi, danio)
31     zombi.recibir_danio!(danio)
32     @energia *= 0.95
33   end
34 end
```

► Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  Mumuki (<http://mumuki.org/>)





Camiones muy especiales

Una empresa que se dedica al transporte de cargas en todo el país necesita un programa que le permita gestionar los viajes de sus camiones.

Para sus operaciones habituales cuenta con los siguientes tipos de camiones:

- *Camiones de frutas y verduras* 🚛: estos modernos camiones tienen un nivel de deterioro inicial igual a 0 y su carga inicial de nafta es de 200 litros.
- *Camiones cerealeros* 🚜: son muy viejos y tienen un nivel de deterioro inicial de 10 para todos los camiones. Su carga de nafta inicial es de 120 litros.

Cada vez que realizan un viaje, transportando una carga expresada en kilogramos, ambos tipos de camiones disminuyen su carga de nafta a la mitad y además:

- el nivel de deterioro de los camiones cerealeros aumenta tanto como los kilogramos que transportan.
- el nivel de deterioro de los camiones de frutas y verduras aumenta en 1.

¿Y qué tienen de especiales estos camiones? 🤔 Aunque como siempre podés enviar tu solución las veces que quieras, no la vamos a evaluar automáticamente por lo que **el ejercicio quedará en color celeste**. 😊 Si querés verla en funcionamiento, ¡te invitamos a que la pruebes en la consola!

Definí los objetos, clases y métodos necesarios para que la empresa pueda gestionar sus transportes.

☒ Solución ☒ Consola

```
1 class Camiones
2   def nafta
3     @nafta
4   end
5   def viajar!(kilogramos)
6     @nafta = @nafta/2
7   end
8   def deterioro
9     @deterioro
10  end
11 end
12
13 class CamionesDeFrutasYVerduras < Camiones
14   def initialize
15     @nafta = 200
16     @deterioro = 0
17   end
18   def viajar!(kilogramos)
19     super + @deterioro += 1
20   end
21 end
22
23 class CamionesCerealeros < Camiones
24   def initialize
25     @nafta = 120
26     @deterioro = 10
27   end
28   def viajar!(kilogramos)
29     super + @deterioro += kilogramos
30   end
31 end
32
```

¡Gracias por enviar tu solución!

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  Mumuki (<http://mumuki.org/>)





¡Sin energía!

Recordemos a nuestra vieja amiga `pepita` y su método `volar_en_circulos!`. Ahora evolucionó y es un objeto de la clase `Golondrina`:

```
class Golondrina
  def initialize
    @energia = 50
  end

  def energia
    @energia
  end

  def volar_en_circulos!
    @energia -= 20
  end
end

pepita = Golondrina.new
```

¿Qué le pasará cuando vuele *demasiado*?

En la consola probá lo siguiente:

- Qué pasa con su energía a medida que la hacemos volar en círculos. ¿Hasta qué punto disminuye?
- ¿Puede seguir volando cuando ya no tenga energía?

Consola Biblioteca

```
pepita = Golondrina.new
=> #<Golondrina:0x0000557b4fa92f98 @energia=50>
pepita.volar_en_circulos!
=> 30
pepita.volar_en_circulos!
=> 10
pepita.volar_en_circulos!
=> -10
```



Sólo Volar Si...

No es muy sorprendente: si `pepita` vuela muchas veces, se va a quedar sin energía. Y eventualmente no sólo se volverá negativa, sino que continuará consumiendo energía al volar.

```
pepita.volar_en_circulos! # su energía queda en 30
pepita.volar_en_circulos! # su energía queda en 10
pepita.volar_en_circulos! # su energía queda en -10
pepita.volar_en_circulos! # su energía queda en -20
# etc...
```

Si bien es fácil de entender, esto está claramente mal: la energía de `pepita` debería ser siempre positiva. Y no debería hacer actividades que le consuman más energía de la que tiene. ¿Qué podríamos hacer?

Modificá el método `volar_en_circulos!` para que sólo vuela (pierda energía) si puede.

☒ ¡Dame una pista!

Una Golondrina puede volar cuando su energía es suficiente, es decir, `@energia >= 20`.

Recordá que además de envíos de mensajes, en objetos contamos con la alternativa condicional o `if`. ¡Refresquemos cómo era su sintaxis! ☺

```
if dia.es_soleado?
  picnic.preparar!
end
```

☒ Solución ☒ Consola

```
1 class Golondrina
2   def initialize
3     @energia = 50
4   end
5
6   def energia
7     @energia
8   end
9
10  def volar_en_circulos!
11    if @energia >= 20
12      @energia -= 20
13    end
14  end
15 end
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas

Lo que acabamos de hacer *funciona*, pero tiene un problema importante: le estamos diciendo a `pepita` que vuela en círculos, pero ahora no sabemos realmente si lo hizo o no.

Esto es lo que se conoce como una *falla silenciosa*: no hizo lo que debía, ¡pero no nos avisó que falló! ☹



Una falla silenciosa

Que los objetos *fallen silenciosamente* es malo porque perdemos la confianza en ellos 😢: no estamos seguros de que el objeto haya cumplido con nuestra orden.

Esto no parece tan terrible cuando del vuelo de las golondrinas se trata, pero ¿y si estamos haciendo una transferencia bancaria?

```
class Transferencia
  def initialize(monto_a_transferir)
    @monto = monto_a_transferir
  end

  def realizar!(origen, destino)
    origen.debitar! @monto
    destino.depositar! @monto
  end
end

transferencia = Transferencia.new(40)
cuenta_origen = CuentaOrigen.new
cuenta_destino = CuentaDestino.new
```

¿Qué sucedería si realizamos la transferencia y `debitar!` no debitara de la cuenta origen cuando no tiene saldo?

¡Descubrilo! Haciendo consultas en la consola, averiguá con cuánto dinero comienzan y terminan la `cuenta_origen` y la `cuenta_destino`.

Asumí que ambas cuentas entienden el mensaje `saldo`.

☒ ¡Dame una pista!

Si no estás seguro de qué probar, te hacemos una seguridad:

1. ☒ `cuenta_origen.saldo`
2. ☒ `cuenta_destino.saldo`
3. ☒ `transferencia.realizar!(cuenta_origen, cuenta_destino)`
4. ☒ `cuenta_origen.saldo`
5. ☒ `cuenta_destino.saldo`

☒ Consola ☒ Biblioteca

```
☒ cuenta_origen.saldo
=> 20
☒ cuenta_destino.saldo
=> 100
☒ transferencia.realizar!(cuenta_origen, cuenta_destino)
=> 140
☒ cuenta_origen.saldo
=> 20
```



¡Fallar!

En el ejemplo que acabamos de ver, si la cuenta origen no tiene suficiente saldo, cuando hagamos `transferencia.realizar!`, de `cuenta_origen` no se habrá debitado nada, pero en la de destino se habrá acreditado dinero. ¡Acabamos de crear dinero! 💰

Suena divertido, pero el banco estará furioso 😠.

El problema acá surge porque la cuenta origen falló, pero lo hizo en silencio y nadie se enteró. ¿La solución? ¡Gritar el error fuerte y claro!

Probá nuevamente las consultas anteriores, pero con una nueva versión del código que **no falla silenciosamente**:

```
☒ cuenta_origen.saldo  
☒ cuenta_destino.saldo  
☒ transferencia.realizar!(cuenta_origen, cuenta_destino)
```

☒ **Consola** ☒ Biblioteca

```
☒ cuenta_origen.saldo  
=> 20  
☒ cuenta_destino.saldo  
=> 100  
☒ transferencia.realizar!(cuenta_origen, cuenta_destino)  
No se puede debitar, porque el monto $40 es mayor al saldo $20 (RuntimeError)  
☒
```

```
class Transferencia
  def initialize(monto_a_transferir)
    @monto = monto_a_transferir
  end

  def realizar!(origen, destino)
    origen.debitar! @monto
    destino.depositar! @monto
  end
end

class CuentaOrigen
  def initialize
    @saldo = 20
  end

  def debitar!(monto)
    if monto > @saldo
      raise "No se puede debitar, porque el monto #{monto} es mayor al saldo #{@saldo}"
    end
  end

  def saldo
    @saldo
  end
end

class CuentaDestino
  def initialize
    @saldo = 100
  end

  def depositar!(monto)
    @saldo += monto
  end

  def saldo
    @saldo
  end
end

transferencia = Transferencia.new(40)
cuenta_origen = CuentaOrigen.new
cuenta_destino = CuentaDestino.new
```



Lanzando excepciones

¿Interesante, no? No solamente tuvimos un mensaje de error claro que nos permite entender qué sucedió, sino que además evitó que se depositase dinero en la cuenta de destino 😅. ¿Cómo fue esto posible?

La primera versión del método `debitar!` en `CuentaOrigen` se veía aproximadamente así:

```
def debitar!(monto)
  if monto <= @saldo
    @saldo -= monto
  end
end
```

Pero la segunda versión se ve así:

```
def debitar!(monto)
  if monto > @saldo
    raise "No se puede debitar, porque el monto ${monto} es mayor al saldo ${@saldo}"
  end

  @saldo -= monto
end
```

Mediante la sentencia `raise` `mensaje` lo que hicimos fue *lanzar una excepción*: provocar un error explícito que *interrumpe* el flujo del programa.

¡Más despacio cerebrito! 🧠 Probá enviar `mensaje_raro` a `ObjetoRaro` (que ya cargamos por vos) en la consola...

```
module ObjetoRaro
  def self.mensaje_raro
    raise "foo"
    4
  end
end
```

...y pensá: ¿se retorna el 4? ¿Por qué?

☒ Consola ☒ Biblioteca

```
☒ ObjetoRaro.mensaje_raro
foo (RuntimeError)
```



Abortando la evaluación

Cuando lanzamos una excepción mediante `raise` mensaje estamos abortando la evaluación del método: a partir de ese momento todas las sentencias que faltaba evaluar serán ignoradas. ¡Ni siquiera llega a retornar nada!

Veamos si va quedando claro: modifiquemos a `Golondrina` para que, en caso de no poder volar, no falle silenciosamente sino que lance una excepción. El mensaje debe ser "No tengo suficiente energía".

Solución Consola

```
1 class Golondrina
2   def initialize
3     @energia = 50
4   end
5
6   def energia
7     @energia
8   end
9
10  def volar_en_circulos!
11    if @energia <= 20
12      raise "No tengo suficiente energía"
13    end
14    @energia -= 20
15  end
16 end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¡Bien hecho! 😊

Sin embargo, las excepciones hacen más que sólo impedir que el resto del método se evalúe, sino que, cuando se lanzan, pueden abortar también la evaluación de todos los métodos de la cadena de envío de mensajes. Veamos por qué...



Paren todo

Como decíamos recién, las excepciones no abortan simplemente la evaluación del método, sino que también abortan la evaluación de toda la cadena de envío de mensajes.

Por ejemplo, si bien en el programa anterior `CuentaOrigen.debitar!(monto)` era un mensaje que podía lanzar una excepción....

```
defdebitar!(monto)
  if monto > @saldo
    raise "No se puede debitar, porque el monto ${monto} es mayor al saldo ${@saldo}"
  end

  @saldo -= monto
end
```

...esta excepción no sólo evitaba que se evaluara `saldo -= monto`, sino que también evitaba que `CuentaDestino.depositar! monto` se enviara. Mirá el código de `realizar!` en Transferencia:

```
defrealizar!(origen, destino)
  origen.debitar! @monto
  destino.depositar! @monto
end
```

A esto nos referimos cuando decimos que las excepciones interrumpen el flujo del programa 😊.

Veamos si se entiende: agregá a la clase `Transferencia` un método `deshacer!` que sea exactamente al revés del `realizar!`: debe revertir la transferencia, moviendo el monto de la cuenta destino a la de origen.

Como ahora tanto la cuenta origen como la cuenta destino pueden `debitar` y `depositar`, unificamos su comportamiento en una clase `Cuenta`. La podés ver en la solapa Biblioteca.

☒ Solución ☒ Biblioteca ☒ Consola

```
1 class Transferencia
2   def initialize(monto_a_transferir)
3     @monto = monto_a_transferir
4   end
5
6   def realizar!(origen, destino)
7     origen.debitar! @monto
8     destino.depositar! @monto
9   end
10
11  def deshacer!(origen, destino)
12    destino.debitar! @monto
13    origen.depositar! @monto
14  end
15 end
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas

¿Será lo mismo debitar y luego depositar que depositar y luego debitar? 🤔

```
class Cuenta
  def initialize(saldo_inicial)
    @saldo = saldo_inicial
  end

  def debitar!(monto)
    if monto > @saldo
      raise "No se puede debitar, porque el monto ${monto} es mayor al saldo ${@saldo}"
    end

    @saldo -= monto
  end

  def depositar!(monto)
    @saldo += monto
  end

  def saldo
    @saldo
  end
end

cuenta_origen = Cuenta.new(20)
cuenta_destino = Cuenta.new(100)
```



El orden importa

Cuando trabajamos con excepciones el orden es importante: lanzar una excepción interrumpe el flujo de ejecución a partir de ese momento, pero no descarta cambios realizados anteriormente: lo que pasó, pasó.

Por eso, como regla práctica, siempre que queremos validar alguna situación, lo haremos siempre antes de realizar las operaciones con efecto . Por ejemplo:

- si vamos a cocinar, vamos a verificar que contemos con todos los ingredientes antes de prender las sartenes
- si vamos a bailar, nos aseguraremos de que contemos con el calzado adecuado antes de entrar en la pista de baile

Veamos si queda claro: éste código tienen un problema relativo al manejo de excepciones. ¡Corregilo!

Solución Consola

```
1 class Saqueo
2   def initialize(barco_saqueador)
3     @barco = barco_saqueador
4   end
5
6   def realizar_contra!(ciudad)
7     if (ciudad.puede_hacerle_frente_a?(@barco))
8       raise "No se puede invadir la ciudad"
9     end
10    @barco.preparar_tripulacion!
11    @barco.desembarcar!(ciudad)
12  end
13 end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



Estudiando a pepita

Ahora te toca a vos: un `Ornitologo` investiga el comportamiento de las golondrinas, en particular `pepita`, y como parte de su estudio la hace:

1. `comer_alpiste! 10`
2. `volar_en_circulos! dos veces`
3. `finalmente comer_alpiste! 10.`

Queremos que `Ornitologo` entienda un mensaje `estudiar_pepita!` que le haga hacer su rutina y que lance una excepción si `pepita.volar_en_circulos!` la lanza.

Escribí el código necesario y pensá si es necesario hacer algo especial para que la excepción que lanza `Pepita` se lance también en `estudiar_pepita!`.

Solución Biblioteca Consola

```
1 class Ornitologo
2   def estudiar_pepita!
3     Pepita.comer_alpiste! 10
4     2.times { Pepita.volar_en_circulos! }
5     Pepita.comer_alpiste! 10
6   end
7 end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Moraleja: si

1. `ClaseA` le envía un mensaje `mensaje1` a `ClaseB`.
2. Dentro del método de `mensaje1`, `ClaseB` le envía un mensaje `mensaje2` a `ClaseC`.
3. `mensaje2` lanza una excepción.
4. Se aborta el método `m2` y propaga la excepción, es decir, la excepción continúa a través de la cadena de envío de mensajes que nos llevó hasta ahí.
5. Se aborta también el método `m1`.

Esto significa que las excepciones se propagan automáticamente a quien hayan enviado el mensaje que la provocó. ¡No hay que hacer nada para que eso suceda! 🎉



Un buen mensaje

Es bastante evidente que cuando lanzás una excepción tenés que darle un *mensaje*. Lo que no es evidente es que:

- Por un lado, el mensaje tiene que ser claro y representativo del error. Por ejemplo, el mensaje "ups" no nos dice mucho, mientras que el mensaje "el violín no está afinado" nos da una idea mucho más precisa de qué sucedió;
- y por otro lado, el mensaje está *destinado al programador*: probablemente el usuario final que use nuestro sistema de cuentas bancarias probablemente no vea nuestros mensajes de error, sino pantallas mucho más bonitas 🎤. Por el contrario, quien verá estos mensajes será el propio programador, cuando haya cometido algún error.

Por ese motivo, siempre procurá lanzar excepciones con mensajes de error descriptivos. 📄

Veamos si queda claro: este código tiene un problema relativo al manejo de excepciones. ¡Corregílo!

☒ Solución ☒ Consola

```
1 class Golondrina
2   def initialize
3     @energia = 50
4   end
5
6   def energia
7     @energia
8   end
9
10  def comer_alpiste!(cantidad)
11    if cantidad <= 0
12      raise "error! cantidad tiene que ser mayor a 0"
13    end
14    @energia += cantidad * 2
15  end
16 end
17
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas