

Objetos

Objetos y mensajes



La programación con objetos, a veces también llamada *orientada a objetos*, es quizás el paradigma de programación más utilizado hoy en día para construir software. Y parte de una idea muy simple: podemos representar al mundo como un **conjunto de objetos** que *interactúan* entre ellos.

Ahora bien: ¿qué es un objeto? Un objeto es ... ¡una cosa! (*;braaaaavo!* 😊). Bueno, bueno, no cualquier cosa, sino algo que es capaz de hacer alguna tarea, la cual conocemos como **responsabilidad** del objeto.

¿Y cómo interactúan estos objetos? ¿Cómo se ven? ¿Cómo hablan entre ellos?

¡Cuántas preguntas! Vamos por partes: ¡conozcamos a nuestros primeros objetos!

•

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
↳ "holo" == "chau"
```

```
↳ Pepita == Norita
=> false
↳ Pepita == Norita
=> false
↳ Norita == Norita
=> true
↳
```

Siguiente Ejercicio: 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!
```

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

Siguiente Ejercicio: Mensajes, segunda parte ➔

💡Dame una pista!

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

¡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?
```

Siguiente Ejercicio: Un poco de sintaxis ➤

colaboración

¡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!

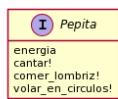
Siguiente Ejercicio: Interfaz ➤

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.

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

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

Siguiente Ejercicio: Hacer versus Devolver ➤

```
↳
```

Siguiente Ejercicio: Hacer versus Devolver ➤

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

Siguiente Ejercicio: 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 `energía` 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 `energía` siempre lo mismo? 🤔

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

```
↳ Pepita.comer_lombriz!
=> nil
↳ Pepita.energia
=> 120
↳ Pepita.volar_en_circulos!
=> nil
↳ Pepita.energia
=> 110
```

Siguiente Ejercicio: 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`

✓ ¡Muy bien! Tu solución pasó todas las pruebas



Podemos sacar dos conclusiones:

1. Los objetos no reaccionan necesariamente 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!

☒ Solución ➤_Consola

```
1 Pepita.comer_lombriz!
2 Pepita.volar_en_circulos!
3 Pepita.comer_lombriz!
4 Pepita.volar_en_circulos!
5 Pepita.comer_lombriz!
6 Pepita.volar_en_circulos!
7 Pepita.comer_lombriz!
8 Pepita.volar_en_circulos!
9 Pepita.comer_lombriz!
10 Pepita.volar_en_circulos!
11
```

▶ Enviar

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

¡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 priiip"

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.

Veamos si queda claro, siendo que las interfaces de [Norita](#), [Pepita](#) y [Mercedes](#) son las siguientes:



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
14 # ¿Qué interfaz comparten Mercedes, Norita y Pepita?
15 interfaz_compartida_entre_todas = %w(
16   cantar!
17 )
```

▶ Enviar

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

Completa 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).

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)
```

Como ves, si envías 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, vuela a `Iruya`, y finalmente vuelva a `Obera`.

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

Siguiente Ejercicio: Más argumentos >

Solución [» Consola](#)

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

▶ Enviar

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. Vuela 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!

Solución [» Consola](#)

```
1 Pepita.comer_alpiste!(90)
2 Pepita.volar_hacia!(Iruya)
3 Pepita.comer_alpiste!((Pepita.energia) * 0.1)
4
```

💡 ¡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`.

✓ ¡Muy bien! Tu solución pasó todas las pruebas

¡Perfecto! 🎉

Un detalle: en Ruby, *a veces*, los paréntesis son opcionales. 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)`).

Es fácil ver que en `Pepita.volar_hacia!(Barreal)` el objeto receptor es `Pepita`, el mensaje `volar_hacia!` y el argumento `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!

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

Siguiente Ejercicio: Recapitulando ➤

Probá en la consola los siguientes envíos de mensajes:

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

Tipos de Mensajes

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!

Solución ➔ Consola

```
1 Pepita.comer_alpiste!(90)
2 Pepita.volar_hacial(Iruya)
3 Pepita.comer_alpiste!((Pepita.energia) * 0.1)
4
```



¡Terminaste Objetos y mensajes!

¡Felicitaciones! Diste tus primeros pasos en el mundo de los objetos. Ya sabés:

- Que un objeto es la **representación** de algún aspecto del problema que queremos resolver. 😕
- Que los objetos tienen **identidad** y saben diferenciarse de otros objetos. 🧑
- Que interactuamos con los objetos mediante el **envío de mensajes**. 📡
- Que al conjunto de mensajes que un objeto entiende lo llamamos **interfaz**. 📄
- Que si le enviamos un mensaje que no entiende, se produce un **error**. 💥

Pero aún queda mucho más por aprender. ¡Te esperamos en la próxima lección!

Crear objetos

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 quisieramos **declarar** a [Norita](#), escribiríramos el siguiente código:

```
module Norita
end
```

Sí, así de simple. 😊

¿Te animás a modificar nuestro código para crear a [Pepita](#)?

▶ Enviar

Solución ➔ Consola

```
1 module Pepita
2 end
```



¡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`.

Definir métodos

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!

✓ ¡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, envía algunos de los mensajes en la **Consola** y fíjate qué **efecto** producen sobre nuestra golondrina.

Solución

Consola

```
↳ Pepita.cantar!
=> nil
↳ Pepita.comer_lombriz!
=> nil
↳ Pepita.volar_en_circulos!
=> nil
↳
```

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.

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! 😊

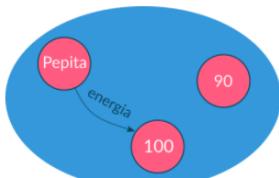
`+ =` y `- =`

Miremos este método con más detenimiento:

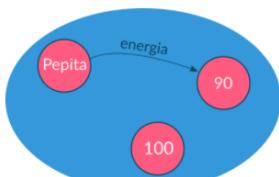
```
def volar_en_circulos!
  @energia = @energia - 10
end
```

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`.

Reescribí 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   def self.comer_lombriz!
8     @energia += 20
9   end
10
11 end
```

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!
  @piata -= 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.

Solución </> Biblioteca > Consola

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

▶ Enviar

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`;

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)
↳
```

Conjunto de atributos: Estados

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
```

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
10 estado_rolamotoC115 = %w(
11 )
12
13
14 estado_enrique = %w(
15   celular
16   dinero_en_billetera
17   frase_favorita
18 )
```

Enviar

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 `energía`, 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   def self.cantar!
9     'pri pri pri'
10  end
11  def self.comer_lombriz!
12    @energia += 20
13  end
14  def self.volar_en_circulos!
15    @energia -= 10
16  end
17  def self.volar_hacia!(destino)
18    @energia -= 100
19    @ciudad = destino
20  end
21  def self.ciudad
22    @ciudad
23  end |
24 end

```

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 kilómetro 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

▶ Enviar

Solución > Consola

```

1 module BuenosAires
2   def self.kilometro
3     0
4   end
5 end
6 module Obera
7   def self.kilometro
8     1040
9   end
10 end
11 module Iruya
12   def self.kilometro
13     1710
14   end
15 end
16 module Pepita
17   @energia = 1000
18   @ciudad = Obera
19   @ciudad1=Iruya
20   def self.energia
21     @energia
22   end

```

```

23   def self.ciudad
24     @ciudad
25   end
26   def self.cantar!
27     'pri pri pri'
28   end
29   def self.comer_lombriz!
30     @energia += 20
31   end
32   def self.volar_en_circulos!
33     @energia -= 10
34   end
35   def self.volar_hacia!(destino)
36     @energia -= (@ciudad.kilometro-
37     destino.kilometro).abs/2
38     @ciudad = destino
39   end

```

Self

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

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

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

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

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

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

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

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

```
1 module Pepita
2   @energia = 1000
3   @ciudad = Obera
4   @distancia=0
5
6
7   def self.energia
8     @energia
9   end
10
11  def self.ciudad
12    @ciudad
13  end
14
15  def self.cantar!
16    'pri pri pri'
17  end
18
19  def self.comer_lombriz!
20    @energia += 20
21  end
22
23  -- def self.volar_en_circulos!
24    @energia -= 10
25  end
26
27  def self.volar_hacia!(destino)
28    self.gastar_energia!(destino)
29
30    @ciudad = destino
31  end
32
33  def self.gastar_energia!(destino)
34    @energia -= (@ciudad.kilometro - destino.kilometro).abs / 2
35    self.distancia_a(destino)
36  end
37
38  def self.distancia_a(destino)
39    (@ciudad.kilometro - destino.kilometro).abs
40  end
41
42 end
```

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.

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!

```
1 module Obera
2
3   def self.kilometro
4     1040
5   end
6   def self.distancia_a(destino)
7     ( destino.kilometro - self.kilometro ).abs
8   end
9 end
10
11 module Iruya
12
13   def self.kilometro
14     1710
15   end
16   def self.distancia_a(destino)
17     ( destino.kilometro-self.kilometro ).abs
18   end
19 end
20
21 module BuenosAires
22
23   def self.kilometro
24     0
25   end
26
27   def self.distancia_a(destino)
28     ( destino.kilometro-self.kilometro ).abs
29   end
30 end
31
32 module Pepita
33   @energia = 1000
34   @ciudad = Obera
35
36
37   def self.energia
38     @energia
39   end
40
41   def self.ciudad
42     @ciudad
43   end
44
45
46   def self.cantar!
47     'pri pri pri'
48   end
49
50   def self.comer_lombriz!
51     @energia += 20
52   end
53
54   def self.volar_en_circulos!
55     @energia -= 10
56   end
57
58   def self.volar_hacia!(destino)
59     self.gastar_energia!(destino)
60     @ciudad = destino
61   end
62
63
64   def self.gastar_energia!(destino)
65     @energia -= @ciudad.distancia_a(destino) / 2
66   end
67
68 end
69
70
```

¡Hola Antonella! Vamos respondiendo por partes.

- Delegar no es en sí usar `self`, sino que la idea de delegar es más o menos lo que veíamos cuando dividiamos el problema en subtareas. En este caso, en el método `gastar_energia!` original se hacía un cálculo de distancias y luego se hacia el cambio en el atributo `@energia`, es decir, se hacían dos cosas, y una de ellas (calcular la distancia) no tenía que ver con el nombre del método, por lo que se hizo un método aparte que realice este cálculo, o sea, se le delegó esta tarea a otro método.
- Siguiendo con eso, el `self` se utiliza cuando dentro de un módulo (o clase) se quiere referir a sí mismo. En este caso el método `distancia_a` era de `Pepita` y se utilizaba en un método de ella misma, por lo que se usaba el `self` para indicar eso. Luego, cuando el método `distancia_a` dejó de estar en `Pepita` y pasó a estar definido en las ciudades ya no se pudo usar el `self` para usar este método, porque como te decía, ya no le pertenecía a `Pepita`. En su lugar hay que usar `@ciudad` ya que este método ahora está definido en las ciudades.
- Por último, lo mismo pasa cuando se cambió `@ciudad` por `self` en el método `distancia_a` cuando lo sacaste de `Pepita` y lo pusiste en cada ciudad. `@ciudad` no es un atributo que tengan las ciudades, ya que ellas mismas son ciudades, por lo que si le dejamos `@ciudad.kilometro` no lo van a entender, la idea es que el cálculo sea al diferencia en los kilómetros de la ciudad destino y ella misma, y para esto tenemos el `self`.

Espero haber podido aclarar, al menos un poco, las dudas. Cualquier cosa si seguís con dudas podés volver a consultar y lo seguimos viendo.

Con lo que aprendiste en esta lección, ya podés crear un **programa** completo, creando los objetos que te sean necesarios. 

Vimos además dos de los conceptos más poderosos del paradigma: la **delegación** y la **distribución de responsabilidades**. Estas herramientas son el principio básico para la **división en subtareas** y seguirán acompañándonos en todas las lecciones.

Polimorfismo y encapsulamiento

Ya que te vas familiarizando con este paradigma, aprovecharemos para hacer algunas reflexiones sobre cómo los objetos interactúan entre sí.

A partir de los conceptos que ya conocés, veremos algunas herramientas conceptuales que nos ayudarán a contestar algunas preguntas:

- ¿Puede un objeto interactuar con otros sin saber quiénes son?
- Si quiero poder hacer algo con dos objetos distintos ¿necesito que entiendan exactamente los mismos mensajes?
- ¿Se puede modificar el **estado** de un objeto "desde afuera"?

Esto, y algunas cosas más sobre aves, entrenamientos y gauchos, en esta lección. ¡Empecemos!

¿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.

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   @energia < 100
18 end
19
20 def self.feliz?
21   @energia > 1000
22 end
```

22 ena
23 end

▶ Enviar

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. 😊

Alternativa convencional if

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   @energia < 100
18 end
19
20 def self.feliz?
21   @energia > 1000
22 end
23
24 def self.hacer_lo_que_quiera!
25
26   if self.debil?
27     self.comer_alpiste!(10)
28   end
29 end
30 end
31
32
```

Times , if y else

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

```
module Pepita
  @energia = 1000
  def self.energia
    @energia
  end
  def self.volar_en_circulos!
    @energia -= 10
  end
  def self.comer_alpiste!(gramos)
    @energia += gramos * 15
  end
  def self.debil?
    @energia<100
  end
  def self.feliz?
    @energia>1000
  end
  def self.hacer_lo_que_quiera!
    if self.debil?
      self.comer_alpiste!(10)
    else
      3.times {self.volar_en_circulos!}
    end
  end
end
```

Condiciones anidadas

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.notaconceptual (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 si feliz, vuela en círculos cinco veces.

Solución > Consola

```
module Pepita
  @energia = 1000
  def self.energia
    @energia
  end
  def self.volar_en_circulos!
    @energia -= 10
  end
  def self.comer_alpiste!(gramos)
    @energia += gramos * 15
  end
  def self.debil?
    @energia<100
  end
end
```

- Si no está feliz ni débil, vuela en círculos 3 veces.

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

```

21  def self.feliz?
22  @energia>1000
23  end
24
25
26
27
28  def self.hacer_lo_que_quiera!
29
30  if self.debil?
31
32    self.comer_alpiste!(10)
33
34  else
35
36  if self.feliz?
37
38    5.times {self.volar_en_circulos!}
39
40  else
41
42    3.times {self.volar_en_circulos!}
43
44  end
45  end
46 end
47 end
48
49
50

```

Elsif

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

```

```

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`? 😊

;Buenas Camila!

Esto es porque `if elif else` es una estructura que vamos a querer agrupar como tal (como una sola), pero si hacemos `else if` estamos haciendo un `if` dentro de otro:

- Caso `elif`:

```
if(condicion1)
...
elif (condicion2)
...
else
...
end #cierro la estructura entera
```

- Caso `else if`:

```
if(condicion1) #abro una estructura de if1
...
else #else del if1
    if(condicion2) #abro una estructura de if2 dentro del caso else del primer if (if1)
    ...
else #else del if2
    ...
end #cierro la estructura del if2 que sigue estando dentro del else del primer if (if1)
end #cierro la estructura del if1
```

Podemos pensarlo como "un end por cada if", en el primera caso tenemos un solo `if` (el `elif` no es un `if`, es su propio caso), en el segundo caso tenemos dos `if`, por ende tenemos que cerrar cada uno de estos y tenemos dos `end`.

El `end` extra que aparece luego de estos es el `end` del método, recordemos que cada método tiene su propio `end` porque también es una estructura por su parte:

```
def self.metodo
...
end
```

Si dentro de este tenemos un `if`, vamos a respetar lo que dijimos arriba, si tenemos un solo if:

```
def self.metodo
    if(condicion1)
    ...
    elif (condicion2)
    ...
    else
    ...
end #cierro la estructura entera
end #cierro el método
```

Si tenemos dos if:

```
def self.metodo
    if(condicion1) #abro una estructura de if1
    ...
    else #else del if1
        if(condicion2) #abro una estructura de if2 dentro del caso else del primer if (if1)
        ...
        else #else del if2
        ...
    end #cierro la estructura del if2 que sigue estando dentro del else del primer if (if1)
    end #cierro la estructura del if1
end #cierro el método
```

Espero que sea de ayuda, si no lo volvemos a ver 😊

```

28 def self.hacer_lo_que_quiera!
29
30   if self.debil?
31     self.comer_alpiste!(10)
32   elsif self.feliz?
33     5.times {self.volar_en_circulos!}
34   else
35     3.times {self.volar_en_circulos!}
36   end
37 end
38
39 end
40
41 end
42
43 end
44 end
45
46

```

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!

Solución

```

1 module Pepo
2   @energia = 1000
3
4
5   def self.energia
6     @energia
7   end
8
9   def self.volar_en_circulos!
10
11   if self.pesado?
12
13     @energia -= 15
14
15   else
16
17     @energia -= 5
18
19   end
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37

```

```

22   def self.comer_alpiste!(gramos)
23     @energia += gramos/2
24   end
25
26   def self.pesado?
27     @energia > 1100
28   end
29
30   def self.hacer_lo_que_quiera!
31
32     self.comer_alpiste!(120)
33
34   end
35
36
37

```

↳ Pepo
=> Pepo
↳ Pepo.hacer_lo_que_quiera!
=> 1060

↳ Pepo.comer_alpiste!(200)
=> 1160
↳ Pepo.volar_en_círculos!
=> 1145
↳ Pepo.hacer_lo_que_quiera!
=> 1205

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!

Solución </> Biblioteca > Consola

```
1 module Pachorra
2
3
4 def self.entrenar_ave!
5
6 10.times { Pepita.volar_en_circulos! }
7
8 Pepita.comer_alpiste!(30)
9
10 5.times { Pepita.volar_en_circulos! }
11
12 Pepita.hacer_lo_que_quiera!
13
14
15 end
16
17 end
18
19
20
21
22
```

```
1 module Pachorra
2
3
4
5 def self.firmar_contrato!(nombre)
6   @ave= nombre
7
8 end
9
10 def self.entrenar_ave!
11
12
13 10.times { @ave.volar_en_circulos! }
14
15 @ave.comer_alpiste!(30)
16
17 5.times { @ave.volar_en_circulos! }
18
19 @ave.hacer_lo_que_quiera!
20
21
22 end
23
24 end
25
```

¿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
```

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

Siguiente Ejercicio: Un entrenamiento más duro ➤

Pero, ¿podrá entrenar con [Pachorra](#)? 😊

Probalo en la consola, enviando los siguientes mensajes:

```
↳ Pachorra.firmar_contrato!(Norita)
↳ Pachorra.entrenar_ave!
```

```
↳ 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.volcar_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.

```
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
```

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

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

Polimorfismo

¿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.

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.

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. 😊

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

Solución

```
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
```

Setters

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:

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
```

```
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
```

↳ `Pachorra.ave=Pepita`

=> `Pepita`

↳ `Pachorra.entrenar_ave!`

=> `5`

Setters y getters

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.

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](#), una tortuga viajera. Algunos de sus atributos pueden ser leídos, otros modificados y otros ambas cosas.

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

💡 Dame una pista!

```
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
```

Solución

Biblioteca Consola

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

Enviar

💡 ¡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.

✓ ¡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. 😊

¡Por supuesto!

- El **getter** es el método que nos sirve para **consultar** un atributo. Por ejemplo si tenemos un atributo:

```
module Persona  
  @edad  
end
```

Si queremos saber el valor de ese atributo, es decir queremos saber la edad de la persona, creamos el método **getter** con el mismo nombre:

```
def self.edad  
  @edad  
end
```

Si fuese un booleano, por ejemplo la persona es mayor de edad:

```
module Persona  
  @mayor_de_edad  
end
```

También podemos crear un **getter** pero como es un **booleano** al nombre le agregamos un `??`. Esto puede servir para recordar, que los booleanos son de alguna forma preguntas, por ejemplo: ¿es mayor de edad?, y lo contesto con sí o no. El **getter** de este atributo sería:

```
def self.mayor_de_edad?  
  @mayor_de_edad  
end
```

- El **setter** por otra parte es un método mediante el cual cambiamos el valor del atributo, le asignamos un nuevo valor. Por ejemplo supongamos que una persona tiene un atributo para saber el color de la remera que está usando, pero cada tanto le gusta cambiarsela:

```
module Persona  
  @color_remera  
end
```

Queremos tener un método **setter** que nos permita **cambiar el valor** de `color_remera` mediante un parámetro (que sería el nuevo color).

Creamos también los **setters** con el mismo nombre, pero les agregamos un `=` para diferenciarlos, ya que por dentro estamos usando el `=` para asignar el nuevo valor:

```
def self.color_remera=(nuevo_color)  
  @color_remera = nuevo_color  
end
```

Espero que sirva, cualquier cosa no dudes en volver a consultar 😊

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

Tu tarea será completar el código que te ofrecemos, 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
8 def self.compинче
9   @compинче
10 end
11
12 def self.compинче=(compинче_nuevo)
13   @compинче = compинче_nuevo
14 end
15
16 ...
17
18 module Eulogia
19   @enojada = false
20
21 def self.enojada?
22   @enojada
23 end
24 end
25
26 module Mendieta
27
28   @ganas_de_hablar = 5
29
30 def self.ganas_de_hablar
31   @ganas_de_hablar
32 end
33
34   def self.ganas_de_hablar=(hablar)
35     @ganas_de_hablar = hablar
36   end
37
38 end
39
```

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, `Elogia` 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).

Defín los métodos `tomar_mate!`, en `Inodoro`, y `recibir_mate!` en `Elogia` y `Mendieta`.

💡 ¡Dame una pista!

 Solución > Consola

```
1 module Inodoro
2   @cafeína_en_sangre=90
3
4   def self.tomar_mate!
5     @cafeína_en_sangre+=10
6     @compinche.recibir_mate!
7   end
8
9
10 def self.cafeína_en_sangre
11   @cafeína_en_sangre
12 end
13
14
15 def self.compinche
16   @compinche
17 end
18
19
20 def self.compinche=(nombre)
21   @compinche = nombre
22 end
23
24 end
25
26
27
28 module Elogia
29   @enojada = false
30
31 def self.enojada?
32   @enojada
33 end
34
35
36 def self.recibir_mate!
37   @enojada= true
38 end
39
40
41 module Mendieta
42   @gananas_de_hablar = 5
43
44 def self.gananas_de_hablar
45   @gananas_de_hablar
46 end
47
48   def self.gananas_de_hablar=(hablar)
49     @gananas_de_hablar = hablar
50   end
51
52 def self.recibir_mate!
53   self.gananas_de_hablar=0
54 | end
55 end
56
```

En esta lección le dimos nombre al **polimorfismo** una idea con la que ya venías trabajando, pero sobre la que todavía no habíamos reflexionado. Este principio fundamental del paradigma de objetos nos permite que podamos interactuar de igual manera con diferentes objetos, con el único requisito de que todos ellos entiendan el o los mensajes que necesitamos enviarles.

Relacionado a esto, hablamos del **encapsulamiento** que nos permite el paradigma, haciendo que cada objeto solo **exponga** lo necesario para interactuar con él y se reserve para su ámbito privado lo que no sea necesario compartir.

En el caso de los atributos, esta exposición se logra implementando un **getter** (método que nos permite ver su valor) o un **setter** (método que nos permite modificar su valor). Y que nuestro código sea entendido fácilmente por otras personas, elegimos utilizar una **convención** para darle nombre a estos métodos.

Colecciones

Ya lo sabemos: un objeto es una "cosa" o ente que cumple ciertas responsabilidades. Pero... ¿qué sucede cuando queremos tratar no una sola cosa, sino a varias al mismo tiempo?

¡Conozcamos a las **colecciones**!

¡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!

```
 Solución > Consola
```

```
1 module CarlosDuty
2
3   @cantidad_logros=0
4   @violento=true
5
6   def self.dificultad
7     30 - @cantidad_logros * 0.5
8   end
9
10  def self.violento?
11    @violento=true
12  end
13
14
15  def self.jugar!(un_tiempo)
16    if un_tiempo > 2
17      @cantidad_logros +=1
18    end
19  end
20
21 end
22
```

```
22
23 module TimbaElLeon
24
25   @dificultad= 25
26
27   def self.dificultad
28     @dificultad
29   end
30
31   def self.violento?
32     @violento=false
33   end
34
35   def self.jugar!(un_tiempo)
36     @dificultad += un_tiempo
37   end
38
39 end
40
```

```

41 module Metroide
42   @nivel_espacial=3
43
44   @violento=false
45
46   def self.dificultad
47     100
48   end
49
50   def self.violento?
51     if (@nivel_espacial >5)
52       @violento = true
53     end
54   end
55
56   def self.jugar!(tiempo)
57     @nivel_espacial += 1
58   end
59
60 end
61

```

```

↳ CarlosDuty.dificultad
=> 30.0

↳ CarlosDuty.violento?
=> true

↳ CarlosDuty.jugar!(un_tiempo)

```

```

↳ CarlosDuty.jugar!(2)
=> nil

↳ CarlosDuty.cantidad_logros?
=> 0

↳ CarlosDuty.jugar!(5)
=> 1

↳ CarlosDuty.cantidad_logros?
=> 1

```

```

↳ TimbaElLeon.dificultad
=> 25

↳ TimbaElLeon.violento?
=> false

```

```

↳ TimbaElLeon.jugar!(5)
=> 30

↳ TimbaElLeon.dificultad
=> 30

```

¡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, 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! 🎉

Biblioteca

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!

`push`, `delete`, `include?` y `size`

¡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 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. 😊

```
↳ numeros_de_la_suerte.push 6
=> [6, 42, 9, 6]
↳ numeros_de_la_suerte.delete 9
=> 9
↳ numeros_de_la_suerte
=> [6, 42, 6]
```

```

↳ numeros_de_la_suerte.include?42
=> true
↳ numeros_de_la_suerte.size
=> 3

```

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!

Solución > Consola

```

1 module Biblioteca
2   @juegos = [CarlosDuty,TimbaElLeon, Metroide]
3   @puntos=0
4   @violento=true
5
6
7   def self.juegos
8     @juegos
9   end
10
11  def self.puntos
12    @puntos
13  end
14 end
15
16  def self.adquirir_juego!(un_juego)
17    @juegos.push un_juego
18    @puntos+=150
19 end
20
21  def self.borrar_juego!(un_juego)
22    @juegos.delete un_juego
23  end
24
25
26  def self.completa?
27    (@puntos>1000 && @juegos.size> 5)
28  end
29
30  def self.violento?
31    @violento
32  end
33
34  def self.juego_recomendable?(un_juego)
35    (!@juegos.include? (un_juego) ) &&
36    (un_juego.violento?))
37
38
39

```

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... 😊

Bloques , proc , call

¡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 paso: 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 (no confundir con nuestro objeto `Biblioteca` 😊)!

Solución

◀ Biblioteca ▶ Consola

```
1
2
3 jugar_a_timba = proc { |minutos| TimbaElLeon.jugar!
(minutos/60)}
```

▶ Enviar

- 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.

¡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.

select

¿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]
```

Solución ↳ Biblioteca ➔ Consola

```
1 module Biblioteca
2   @juegos = [CarlosDuty, TimbaElLeon, Metroide]
3
4   def self.juegos
5     @juegos
6   end
7
8
9   def self.juegos_violentos
10    juegos_violentos= @juegos.select
11    {|un_juego|un_juego.violento?}
12  end
13
14
15
16
17
```

▶ Enviar

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!

¡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.

Find

¡Hola Franco! ¿Como andas?

Primero antes que nada vamos a pensar en como funciona el `find`. Este mensaje se mandara a una colección (en este caso `@juegos` ya que es un atributo) y buscara lo que dice dentro de las llaves. Pero ¿Que es lo que hay dentro? Sabemos que una colección puede tener mas de un elemento dentro como lo vemos, lo cual es muy logico. Entonces apenas se abren las llaves le escribiremos `||` y adentro de eso un nombre que los identifique. Esto es mas que nada para que cada elemento se posicione ahí y se vaya corroborando si cumple la condición como para decir que se encontro o no. Por ultimo, luego de eso escribiremos nuestra condición con aquel nombre que sera reemplazado por los elementos.

Medio confuso lo se, pero pensemos con un ejemplo. Tengo una colección llamada `numero = [1, 2, 3, 4]` y queremos ver si encontramos un numero mayor al 3 (si si si, tal como el ejemplo, lo se, me falto creatividad jajaja). Entonces vamos a escribir `colección.find({variable|condición con variable})`, por lo tanto, queda algo asi: `numero.find{|cadaNúmero| cadaNúmero>3}`.

Primer elemento el 1: `cadaNúmero = 1` -----> ¿1 es mayor a 3? No. Perfecto siguiente

Primer elemento el 2: `cadaNúmero = 2` -----> ¿2 es mayor a 3? No. Perfecto siguiente

Primer elemento el 3: `cadaNúmero = 3` -----> ¿3 es mayor a 3? No. Perfecto siguiente

Primer elemento el 4: `cadaNúmero = 4` -----> ¿4 es mayor a 3? SI! Perfecto encontramos uno entonces se devuelve el 4.

Muy bien, ya sabemos como funciona el `find` pero este ya se "notifica" y en ningun momento debemos almacenarlo o guardararlo en algun atributo ya que, por lo menos por ahora, no lo vamos a usar en otro lado, solo queremos informarlo.

Espero que asi se comprenda mejor pero si no es asi, volve a preguntar y te doy una mano.

¿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(uná_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(uná_dificultad)
9     juego_mas_dificil_que=@juegos.find{|un_juego|una_dificultad<un_juego.dificultad}
10  end
11 end
12
13
14
15
16
```

Enviar

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

All? Y any?

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? }
```

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
15  end
16
17
18
19
20
21
22
23
```

¿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`.

Biblioteca.mucha_violencia?

=> false

Biblioteca.muy_dificil?

=> true



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

```
1 module Biblioteca
2   @juegos = [CarlosDuty, TimbaElLeon, Metroide]
3
4   def self.juegos
5     @juegos
6   end
7
8
9   def self.juegos_violentos
10    juegos_violentos = @juegos.select
11    {|un_juego|un_juego.violento?}
12  end
13
14  def self.dificultad_violenta
15    dificulta_violanta = self.juegos_violentos.map
16    {|dificultades| dificultades.dificultad}
17  end
18
19
```

Count y sum

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_deViolencia`, cuyo valor sea la sumatoria de dificultad de los juegos violentos dividida por la cantidad de juegos violentos de la `Biblioteca`.

Solución

```
1 module Biblioteca
2   @juegos = [CarlosDuty, TimbaElLeon, Metroide]
3
4   def self.juegos
5     @juegos
6   end
7
8
9   def self.juegos_violentos
10    juegos_violentos = @juegos.select
11    {|un_juego|un_juego.violento?}
12  end
13
14  def self.cantidad_juegos_violentos
15    juegos_violentos.count{|juego|juego.violento?}
16  end
17
18  def self.promedio_de_violencia
19    juegos_violentos.sum{|juego|juego.dificultad}/cantidad_juegos_violentos
20
21
22
```

Each

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! 😊

Defini 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)`.

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

¡Terminaste Colecciones!

¡Felicitaciones! 🎉 A lo largo de esta guía aprendiste:

- Qué son las colecciones
- Qué mensajes básicos entienden: agregar y quitar elementos mediante `push` y `delete`, preguntar si existe un elemento usando `include?`, y saber el tamaño de la colección enviándole `size`
- Qué son los bloques, cómo se aplican y cómo se les pasan parámetros
- Mensajes más complejos que utilizan bloques como `select`, `find`, `map`, `all?`, `any?`, `count`, `sum` y `each`. ¡Son un montón! 🤪

Referencias

Hasta ahora venimos hablando mucho de objetos y mensajes. Pero ocasionalmente se nos escapó una palabra: *referencia*. ¿Qué es? ¿Cómo se relacionan las referencias con los objetos y los atributos?

Estas y muchas preguntas más las responderemos a continuación 😊

upcase

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  
3 saludo_formal = "buen dia"  
4
```

Enviar

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

Sí y no. Veámos por qué...

Variables , referencias y objetos

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.upcase
```

...le estamos enviando el mensaje `upcase` 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 envíale el mensaje `size()`.

💡 ¡Dame una pista!

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

Solución > Consola

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

¡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!

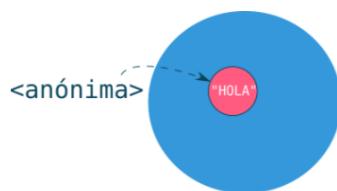
Como vemos, los objetos son las "bolitas" y las referencias, las "flechitas". Pero, ¿cuál es la diferencia entre variable y referencia?

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.abs.even?  
=> true  
↳ (4 + 8).abs  
=> 12
```

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?

```
=> "buen día"  
↳ "buen día".equal? "buen día"  
=> false  
↳ otro_saludo.equal? otro_saludo  
=> true  
↳ despedida.equal? otro_saludo  
=> true  
↳
```

Siguiente Ejercicio: Identidad, revisada ➤

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

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

¡Ahora sacá tus conclusiones! 😊

Equal

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
```

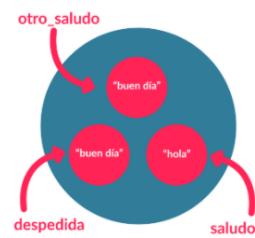
¿Por qué? ¡Simple! Ambas referencias, `otro_saludo` y `despedida`, apuntan al mismo objeto. La moraleja es que declarar una variable significa agregar una nueva referencia al objeto existente, en lugar de copiarlo:



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   pan="harina",  
3   facturas=pan,  
4   tortas=facturas  
5 ]
```

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...

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 referencia a una lista conformada por tres referencias *distintas* que apunten a objetos equivalentes entre sí, pero no idénticos.

⚠ ¡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 lo implementan:
 - Listas
 - Números
 - Strings
 - Booleanos

¿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! 🎂

```
1 module AbuelaClotilde  
2   @felicidad = 100  
3  
4   def self.alimentar_nieto  
5     Fito.comer(2000)  
6     Fito.comer(1000)  
7   end  
8  
9   def self.comer(calorías)  
10    @felicidad += calorías * 0.001  
11  end  
12  
13  def self.felicidad  
14    @felicidad  
15  end  
16 end
```

⚠ 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.

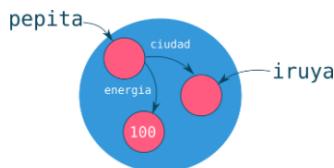
Además de los que ya vimos, hay más tipos de referencias: los atributos.

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:



Solución > Consola

```
1 module Pepita
2   @ciudad
3
4   def self.ciudad=(una_ciudad)
5     @ciudad = una_ciudad
6
7   end
8
9 end
```

Enviar

Nuevamente, acá vemos otro caso de múltiples referencias: el objeto que representa a la ciudad de `Iruya` es globalmente conocido como `Iruya`, y también conocido por `Pepita` como `ciudad`.

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

✓ ¡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.

¿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 a 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?

Solución > Consola

```
1
2
3 Fito.novia=Melisa
4 AbueloGervasio.nieta=Melisa
5
6
7
8 3.times{AbueloGervasio.alimentar_nieta}
9
```

Enviar

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. 😊

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 Jazmin 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)
```

Solución > Consola

```
1
2 module PianoFamiliar
3
4   @nivel_de_afinacion=100
5
6   def self.está_afinado?
7     @nivel_de_afinacion>80
8   end
9
10  def self.afinar!(horas)
11    @nivel_de_afinacion=
12    ([100,@nivel_de_afinacion+horas*5 ].min)
13  end
14
15  def self.desafinar!
16    @nivel_de_afinacion -= 1
17  end
18
19
20 module Jazmin
21   @piano
22
23
24   def self.piano=(nuevo_piano)
25     @piano = nuevo_piano
26   end
27
28   def self.tocar
29     @piano.desafinar!
30   end
31 end
32
33 module Lucio
34
35   def self.afinar(piano,horas)
36     piano.afinar!(horas)
37   end
38 end
```

Enviar

¡Terminaste Referencias!

Repasemos lo recién visto: en un ambiente hay muchos objetos, pero en realidad no interactuamos con ellos directamente, sino a través de referencias, que son nombres o etiquetas que les damos a los objetos.

Para un objeto pueden existir múltiples nombres: cuando le damos uno nuevo, no estamos creando una copia del objeto ni modificándolo realmente, sino que estamos creando una nueva referencia que apunta al objeto. Así que ¡ojo!, si compartís un objeto con otros, y lo mutás, ¡todos los que tengan una referencia al mismo verán los cambios!

Finalmente, en objetos, todo lo que se *parezca* a una variable es una referencia, y hay de muchos tipos:

- variables de un programa
- variables locales de un método
- parámetros de un método
- atributos de un objeto
- y el nombre global de un objeto bien conocido.

Clases e instancias

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 decírnos su `salud`, que inicialmente es 100, pero puede cambiar.

¿Cuánto 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
```

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

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. Declarará un `getter` para este atributo.

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
3   @energia=1000
4
5   def self.energia
6     @energia
7   end
8
9   def self.atacar!(zombi,puntos)
10
11   zombi.recibir_danio!(puntos)
12
13   end
14 end
15
```

▶ Enviar

¿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 Kiki
2
3   @salud=100
4   @correr=false
5
6   def self.sabe_correr?
7     @correr
8   end
9
10  def self.gritar
11    "¡agrrrg!"
12  end
13
14  def self.salud
15    @salud
16  end
17
18  def self.salud=(nueva_salud)
19    @salud=nueva_salud
20  end
21
22  def self.recibir_danio!(puntos)
23    @salud=[@salud-puntos*2,0].max
24  end
25
26  def self.sin_vida?
27    @salud==0
28  end
29
30  end
31
32
33 module Bouba
34
35   @salud=100
36   @correr=false
37
38   def self.sabe_correr?
39     @correr
40   end
41
42   def self.gritar
43    "¡agrrrg!"
44  end
45
46  def self.salud
47    @salud
48  end
```

```

49
50   def self.salud=(nueva_salud)
51     @salud=nueva_salud
52   end
53
54   def self.recibir_danio!(puntos)
55     @salud=[@salud-puntos*2,0].max
56   end
57
58
59   def self.sin_vida?
60     @salud== 0
61   end
62
63 end
64
65 end
66
67

```

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...

Objetos que se comportan exactamente de la misma forma

Class

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 CelularDeMaría
  @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! 🙏

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

▶ Enviar

✅ ¡Muy bien! Tu solución pasó todas las pruebas

💡 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...

Se debe definir en minúsculas

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`.

► Enviar

¿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**, 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, las variables como `saludo`, `despedida`, o `kiki` comienzan con minúscula. 😊

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,5)
uninitialized constant Bouba (NameError)
↳ Juliana.atacar!(bouba,5)
=> 90
↳ Juliana.atacar!(kiki,5)
=> 90
↳ Juliana.atacar!(kiki,7)
=> 76
```

Initialize o inicializar

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 `energia` 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
6   def energia
7     @energia
8   end
9
10  def atacar!(zombi,puntos)
11    zombi.recibir_danio!(puntos)
12  end
13
14  end
15
16 juliana = Sobreviviente.new
17 anastasia =Sobreviviente.new
```

▶ Enviar

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

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

```

23
24 juliana = Sobrevidiente.new
25 anastasia =Sobrevidiente.new
26
27 class Zombi
28   def initialize
29     @salud = 100
30   end
31
32
33 def sabe_correr?
34   @correr=false
35 end
36
37 def gritar
38   "jagrrrg!"
39 end
40
41 def salud
42   @salud
43 end
44
45 def salud=(nueva_salud)
46   @salud=nueva_salud
47 end

48
49 def recibir_danio!(puntos)
50   @salud=[@salud-puntos*2,0].max
51 end
52
53 def sin_vida?
54   @salud==0
55 end
56
57 end
58
59 caminantes = []
60
61
62
63 20.times{caminantes.push(Zombi.new)}
64
65
66

```

¡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`

Initialize con valores que cambian

`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`!

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

Constructor

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)
```

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(numero)
4     @salud = numero
5   end
6
7
8
9   def sabe_correr?
10    @correr=true
11  end
12
13  def gritar
14    "¡agrrrg!"
15  end
16
17  def salud
18    @salud
19  end
20
21  def salud=(nueva_salud)
22    @salud=nueva_salud
23  end
24
25  def recibir_danio!(puntos)
26    @salud=[@salud-puntos*3,0].max
27  end
28
29  def sin_vida?
30    @salud==0
31  end
32
33  def regenerarse!
34    @salud=100
35  end
36
37
38
39 end

```

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. 😊

¡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! 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=@energia/2
17  end
18
19  def beber!
20    @energia=1.25*energia
21  end
22 end

```

¡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.

```
1 class Aliado
2   def initialize
3     @energia = 500
4   end
5
6   def energia
7     @energia
8   end
9
10  def atacar!(zombie, danio)
11    zombie.recibir_danio!(danio)
12    @energia-=@energia*0.05
13  end
14
15  def ataque_masivo!(zombis)
16    zombis.each { |zombi| atacar!(zombi, 20) }
17    @energia=@energia/2
18  end
19
20  def beber!
21    @energia=1.10*energia
22  end
23
24 end
```

¡Terminaste Clases e Instancias!

¡Ya conocemos todo lo necesario acerca de las clases! 🎉

Recordá que sirven para generalizar comportamiento idéntico entre objetos: si no es **exactamente igual**, es necesario crear una nueva clase, como hicimos con `Zombi` y `SuperZombi`, o con `Sobreviviente` y `Aliado`. En esos casos tuvimos que repetir parte de la lógica que coincidía entre las clases; pronto aprenderemos una alternativa. 😊

Herencia

En la guía anterior aprendimos que cuando varios objetos tienen comportamiento en común podemos crear una `clase` que lo agrupe. Allí se define ese comportamiento para evitar la repetición de lógica entre los distintos objetos.

Sin embargo, cuando tenemos clases que tienen una parte de comportamiento común pero otra que difiere, las herramientas que vimos hasta el momento nos quedan cortas.

¡Pero ya no más! ¡Adentrémonos en el mundo de la **herencia**! 😊

¡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 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! 😊

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

¡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
7     def ligero?
8         @litros<10
9     end
10
11    def conducir!(kilometros)
12        @litros-=kilometros*0.1
13    end
14
15    def cantidad_de_ruedas
16        2
17    end
18
19 end
20
```

¿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!
```

Jerarquías

Clases abstractas

💡 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 `Cóndor` 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 querríamos ser tan genéricos con nuestras clases si podemos ser específicos?

En el ejemplo con animales ocurre parecido: si definimos implementaciones específicas para `Cóndor`, `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, `Cóndor` 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 `Cóndor`. Entonces, decimos que `volar!` es un **método abstracto**.

- `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.

Ahora que ya conocemos 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
3   def ligero?
4   end
5
6   def conducir!(kilometros)
7   end
8
9   def cantidad_de_ruedas
10  end
11 end
12|
```

```

13 class Auto<MedioDeTransporte
14
15   def initialize
16     @litros=40
17   end
18
19   def ligero?
20     @litros<20
21   end
22
23   def conducir!(kilometros)
24     @litros-=kilometros*0.05
25   end
26
27   def cantidad_de_ruedas
28     4
29   end
30 end
31
32 class Moto<MedioDeTransporte
33
34   def initialize
35     @litros=20
36   end
37
38   def ligero?
39     @litros<10
40   end
41
42   def conducir!(kilometros)
43     @litros-=kilometros*0.1
44   end
45
46   def cantidad_de_ruedas
47     2
48   end
49 end
50

```

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! 🎉

¡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
3   def ligero?
4   end
5
6   def conducir!(kilometros)
7   end
8
9   def cantidad_de_ruedas
10  end |
11 end
12
13 class Auto<MedioDeTransporte
14
15   def initialize
16     @litros=40
17   end

```

```

19 def ligero?
20   @litros<20
21 end
22
23 def conducir!(kilometros)
24   @litros-=kilometros*0.05
25 end
26
27 def cantidad_de_ruedas
28   4
29 end
30
31 def peso
32   cantidad_de_ruedas*200
33 end
34 end
35
36 class Moto<MedioDeTransporte
37
38   def initialize
39     @litros=20
40   end
41
42   def ligero?
43     @litros<10
44   end
45
46   def conducir!(kilometros)
47     @litros-=kilometros*0.1
48   end
49
50   def cantidad_de_ruedas
51     2
52   end
53
54   def peso
55     cantidad_de_ruedas*200
56   end
57 end
58

```

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!

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

Solución >_Consola

```

1 class MedioDeTransporte
2
3   def ligero?
4   end
5
6   def conducir!(kilometros)
7   end
8
9   def cantidad_de_ruedas
10  end

```

```

12     def peso
13         cantidad_de_ruedas*200
14     end
15
16
17 end
18
19 class Auto<MedioDeTransporte
20
21     def initialize
22         @litros=40
23     end
24
25     def ligero?
26         @litros<20
27     end
28
29     def conducir!(kilometros)
30         @litros-=kilometros*0.05
31     end
32
33     def cantidad_de_ruedas
34         4
35     end
36
37
38 end
39
40 class Moto<MedioDeTransporte
41
42     def initialize
43         @litros=20
44     end
45
46     def ligero?
47         @litros<10
48     end
49
50     def conducir!(kilometros)
51         @litros-=kilometros*0.1
52     end
53
54     def cantidad_de_ruedas
55         2
56     end
57
58
59 end
60

```

¿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! 😊

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

```

▶ Enviar

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

¡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`.

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

Cuando surgieron los `SuperZombi`, 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!`

¡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    "¡agrrrg!"
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
30  def sabe_correr?
31    true
32  end
33
34  def recibir_danio!(puntos)
35    @salud = [@salud - puntos * 3, 0].max
36  end
37
38  def regenerarse!
39    @salud = 100
40  end
41 end
```

⚠ 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 define `Zombi`.

¡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
3   def initialize
4     @litros=100
5     @pasajeros=0
6   end
7
8   def ligero?
9     @pasajeros==0
10  end
11
12  def conducir!(kilometros)
13    @litros-=kilometros*0.2
14  end
15
16  def cantidad_de_ruedas
17    6
18  end
19
20
21 end
22
```

`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! 😊

¿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: agrega los métodos `sube_pasajero!` y `baja_pasajero!` a la clase `Micro`.

■ Solución > _Consola

```
1 class Micro<MedioDeTransporte
2
3   def initialize
4     @litros=100
5     @pasajeros=0
6   end
7
8   def ligero?
9     @pasajeros==0
10  end
11
12  def conducir!(kilometros)
13    @litros-=kilometros*0.2
14  end
15
16  def cantidad_de_ruedas
17    6
18  end
19
20  def sube_pasajero!
21    @pasajeros=@pasajeros+1
22  end
23
24  def baja_pasajero!
25    @pasajeros=@pasajeros-1
26  end
27
28 end
```

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!

Super

¡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
3   def initialize
4     @litros=100
5     @pasajeros=0
6   end
7
8   def ligero?
9     @pasajeros==0
10  end
11
12  def conducir!(kilometros)
13    @litros-=kilometros*0.2
14  end
15
16  def cantidad_de_ruedas
17    6
18  end
19
20  def sube_pasajero!
21    @pasajeros=@pasajeros+1
22  end
23
24  def baja_pasajero!
25    @pasajeros=@pasajeros-1
26  end
27
28  def peso
29    super + @pasajeros*80
30  end
31
32
33
34 end
35
```

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! 🎉

;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

```

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!

```

1 class Persona
2   def initialize
3     @energia = 1000
4   end
5
6   def energia
7     @energia
8   end
9
10  def atacar!(zombi, danio)
11    zombi.recibir_danio!(danio)
12  end
13
14  def beber!
15  end
16
17 end
18
19
20 class Sobrevidiente<Persona
21   def initialize
22     @energia = 1000
23   end
24

```

```

27  def beber!
28    @energia *= 1.25
29  end
30
31  def atacar!(zombi, danio)
32    super
33  end
34 end
35
36 class Aliado<Persona
37   def initialize
38     @energia = 500
39   end
40
41
42  def beber!
43    @energia *= 1.10
44  end
45
46  def atacar!(zombi, danio)
47    super + @energia *= 0.95
48  end
49
50 end

```

¡Ya tenemos una nueva herramienta para evitar repetir lógica! 🎉

Recordá que la **herencia** es un concepto amplio que tiene muchas variantes: primero vimos clases y métodos **abstractos**, que no se instancian y que especifican qué mensajes deben implementar las clases que hereden. También aprendimos a **redefinir** los métodos cuando heredamos un método pero queremos que se comporte de otra forma.

Luego vimos un caso de herencia de una clase concreta, y cómo las subclases **heredan** los métodos de su superclase. Por último aprendimos a usar `super`: cuando una subclase lo envía, se evalúa el método del mismo nombre de su superclase.

Excepciones

¡Hola de nuevo!

Cuando nuestros programas se empiezan a volver más complejos, es frecuente encontrarnos con situaciones en las que los objetos simplemente no pueden hacer lo que les pedimos 😱: correr una carrera sin suficiente nafta, pagar una boleta después de su vencimiento, cantar sin micrófono.

Por más cuidadosos que seamos, es inevitable que ocurran situaciones excepcionales. ¿Y qué podemos hacer ante ellas? ¡En esta lección encontraremos algunas respuestas!

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!

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
16 end
```

▶ Enviar

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?

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`

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)
=> 140
↳ cuenta_origen.saldo
=> 20
```

Siguiente Ejercicio: ¡Fallar!

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)
↳
```

Siguiente Ejercicio: Lanzando excepciones

```
raise
```

¡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 debitarse!(monto)
  if monto <= @saldo
    @saldo -= monto
  end
end
```

Pero la segunda versión se ve así:

```
def debitarse!(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"
  end
end
```

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

► Consola ↗ Biblioteca

↳ ObjetoRaro.mensaje_raro
foo (RuntimeError)

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
12    if @energia >= 20
13      @energia -= 20
14    else @energia < 20
15      raise "No tengo suficiente energía"
16    end
17  end
18
19 end
```

► Enviar

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: modificaremos 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".

```
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
17  end
18 end
19 end
```

¡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...

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

```
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
```

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
13  end
14 end
```

▶ Enviar

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
3   def estudiar_pepita!
4     Pepita.comer_alpiste!(10)
5
6
7   2.times{Pepita.volar_en_circulos!}
8
9
10
11
12   Pepita.comer_alpiste(10)
13
14 end
15
16 end
```

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. ¡Corregilo!

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

▶ Enviar

X

¡Terminaste Excepciones!

¡Genial! 🎉

Aprendimos a lanzar excepciones, a interpretar en qué momento es importante tirarlas, y cómo se propagan a lo largo del envío de mensajes.

Y por supuesto, ¡deben tener mensajes descriptivos!
