Aflojá con el aparatito

Es innegable que en la actualidad los dispositivos electrónicos atraviesan nuestro día a día . Desde celulares hasta *notebooks* que están presentes tanto en nuestro ocio como en nuestros trabajos o estudios. Es por eso que vamos a modelar distintos dispositivos utilizando la programación con objetos.

Para entrar en calor vamos a modelar la clase `Celular` , ¿qué sabemos de ellos?

- Todos los celulares tienen su `@bateria` en 100 inicialmente;
- Cuando utilizamos un `Celular` , su batería disminuye en la mitad de los minutos que lo hagamos. Por ejemplo: si usamos el celular 30 minutos, su batería bajará en 15.
- Los celulares se pueden `cargar_a_tope!` para dejar la batería en 100.

> Veamos si se entiende: definí la clase `Celular` y también los métodos `initialize` , `utilizar!` y `cargar_a_tope!` .

¡Dame una pista!

Solución | Consola

```ruby
class Celular
  def initialize
    @bateria = 100
  end

  def utilizar!(minutos)
    @bateria = [(@bateria-(minutos/2)),0].max
  end

  def cargar_a_tope!
    @bateria = 100
  end

end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¡Excelente!

Pero bien sabemos que no solo utilizamos celulares y que en los últimos años las computadoras portatiles le ganaron terreno a las de escritorio...

Notebook

¡Ahora es el turno de la `Notebook` !

La clase `Notebook` entiende los mismos mensajes que `Celular` y se comporta parecido pero no exactamente igual. La diferencia está en que a la hora de `utilizar!` una notebook, su `@bateria` disminuye en la cantidad de minutos que la utilicemos.

> Definí la clase `Notebook`, que sepa entender los mensajes `initialize`, `utilizar!` y `cargar_a_tope!`.

Solución | Biblioteca | Consola

```
1  class Notebook
2    def initialize
3      @bateria = 100
4    end
5
6    def utilizar!(minutos)
7      @bateria = [(@bateria-minutos),0].max
8    end
9
10   def cargar_a_tope!
11     @bateria = 100
12   end
13
14 end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¡Muy bien! Pero... las clases `Celular` y `Notebook` son demasiado parecidas, ¿no?

Más específicamente en los métodos `initialize` y `cargar_a_tope!` son iguales.

¡Obviamente se puede evitar esa repetición de lógica! Vamos al siguiente ejercicio a ver cómo.

Su superclase

Una forma de organizar las clases cuando programamos en objetos es establecer una **jerarquía**. En nuestro caso podemos pensar que `Celular` y `Notebook` se pueden englobar en algo más grande que las incluya, la idea de `Dispositivo`.

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.

```
class Ave
  def volar!
    @energia -= 20
  end
end

class Condor < Ave
  def dormir!(minutos)
    @energia += minutos * 3
  end
end

class Halcon < Ave
  def dormir!(minutos)
    @energia += minutos
  end
end
```

El símbolo `<` significa "hereda de": por ejemplo, `Condor` hereda de `Ave`, que está *más arriba* en la jerarquía. Otra manera de decirlo es que cada `Condor` es un `Ave`.

La herencia nos permite que las *subclases* (`Condor` y `Halcon`) posean los mismos métodos y atributos que la *superclase* `Ave`. Es decir, las instancias de `Condor` y de `Halcon` van a saber `volar!` de la misma forma, pero cuando les enviemos el mensaje `dormir!` cada una hará algo diferente.

¡Uf! ¡Eso fue un montón! A ver si quedó claro.

> Definí la clase `Dispositivo` y modificá las clases que definiste anteriormente para evitar que haya métodos repetidos entre `Celular` y `Notebook`. Es importante que en el editor definas arriba la superclase y abajo sus subclases.

¡Dame una pista!

Solución | Consola

```ruby
class Dispositivo
  def initialize
    @bateria = 100
  end

  def cargar_a_tope!
    @bateria = 100
  end

end

class Celular < Dispositivo
  def utilizar!(minutos)
    @bateria = [(@bateria-(minutos/2)),0].max
  end

end

class Notebook < Dispositivo
  def utilizar!(minutos)
    @bateria = [(@bateria-minutos),0].max
  end

end


```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¡Genial!

Para recapitular, cuando dos objetos repiten lógica, creamos una clase con el comportamiento en común. En el caso que dos clases repitan lógica deberíamos crear una nueva clase a la cual llamamos superclase. A esta nueva clase llevaremos los métodos repetidos y haremos que las clases originales hereden de ella. Estas subclases que heredan de la superclase solo contendrán su comportamiento particular.

Arte abstracto

Sabiendo que contamos con las clases `Celular` y `Notebook`, ¿alguna vez instanciaremos un objeto de la clase `Dispositivo`? ¡Probablemente no! ¿Por qué querríamos crear algo tan genérico si podemos crear algo más específico?

A este tipo de clases, como `Dispositivo` o `Ave` en el ejemplo del ejercicio anterior, se las llama **clases abstractas** porque, a diferencia de las **clases concretas** (como `Celular` o `Notebook`), nunca las instanciamos. En otras palabras, no creamos objetos con esa clase, solo nos sirven para proveer comportamiento a sus subclases.

¿Me estás cargando?

Una de las grandes molestias que nos traen los dispositivos electrónicos es cuando se quedan sin batería.

Sabemos que tanto los celulares como las notebooks están descargados si tienen 20 o menos de batería.

> Definí el método `descargado?` en donde corresponda.

¡Dame una pista!

Solución | Consola

```ruby

class Dispositivo
  def initialize
    @bateria = 100
  end

  def cargar_a_tope!
    @bateria = 100
  end

  def descargado?
    @bateria <= 20
  end

end

class Celular < Dispositivo
  def utilizar!(minutos)
    @bateria = [(@bateria-(minutos/2)),0].max
  end

end

```

```ruby
class Notebook < Dispositivo
  def utilizar!(minutos)
    @bateria = [(@bateria-minutos),0].max
  end

end

```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¿Acaso para la `Notebook` no deberíamos preguntar si está `descargada?` en lugar de `descargado?` ?

¡Puede ser! Pero si tuvieramos un método por cada clase, no podríamos tratar polimórficamente a los objetos. Por ejemplo, no habría manera de saber cuántos dispositivos están descargados porque no existiría un único mensaje que respondiera nuestra pregunta.

```
> dispositivos.count { |dispositivo| dispositivo.descargado? }
=> #¡Fallaría porque Notebook no podría responder al mensaje descargado?!

> dispositivos.count { |dispositivo| dispositivo.descargada? }
=> #¡Fallaría porque Celular no podría responder al mensaje descargada?!
```

Prueba sorpresa

¡Tranqui! no es la clase de prueba que estás pensando.

Ya hablamos mucho de las ventajas de la herencia para evitar la repetición de lógica. Pero, ¿funciona todo esto que estuvimos haciendo?

Probá en la consola los siguientes comandos:

```
> un_celu = Celular.new
> una_notebook = Notebook.new
> un_celu.descargado?
> un_celu.utilizar! 180
> un_celu.descargado?
> una_notebook.utilizar! 100
> una_notebook.cargar_a_tope!
> una_notebook.descargado?
```

Consola Biblioteca

>

Vamos de paseo

Desconectémonos un poco y salgamos de paseo. ¿En qué vamos?

7. Vamos de paseo

Por ahora nuestras opciones son limitadas. Podemos elegir ir en `Auto` o en `Moto`. De estos medios sabemos que:

- ambos comienzan con una cantidad que podemos establecer de `@combustible`;
- los autos pueden llevar 5 personas como máximo y al `recorrer!` una distancia consumen medio litro de `@combustible` por cada kilómetro recorrido;
- las motos pueden llevar 2 personas y consumen un litro por kilómetro recorrido;
- ambos pueden `cargar_combustible!` en la cantidad que digamos y al hacerlo suben su cantidad de `@combustible`;
- ambos saben responder si `entran?` una cantidad de personas. Esto sucede cuando esa cantidad es menor o igual al máximo que pueden llevar.

¡Vamos a modelar todo esto!

> Definí las clases `Moto`, `Auto` y `MedioDeTransporte` y hace que las dos primeras hereden de la tercera. También definí los métodos `initialize`, `recorrer!`, `cargar_combustible!`, `entran?` y `maximo_personas` donde correspondan.

¡Dame una pista!

Solución | Consola

```ruby
class MedioDeTransporte
  def initialize (combustible)
    @combustible = combustible
  end

  def cargar_combustible!(carga)
    @combustible += carga
  end

  def entran? (cant)
    self.maximo_personas>= cant
  end

end

class Moto < MedioDeTransporte
  def recorrer!(kms)
    @combustible -= kms
  end

  def maximo_personas
    2
  end

end
```

```
26
27 class Auto < MedioDeTransporte
28   def recorrer!(kms)
29     @combustible -= kms/2
30   end
31
32   def maximo_personas
33     5
34   end
35
36 end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¡Excelente!

Estaría bueno tener algún medio de transporte más, ¿no? Acompañanos a la próxima parada.

Subí nomás

¿Y si no tenemos `Auto` ni `Moto`? Vamos a modelar `Colectivo`s así tenemos un poco más de variedad.

Los `Colectivo`s son un `MedioDeTransporte` que tienen un máximo de 20 personas y que al `recorrer!` una distancia gastan el doble de `@combustible` de los kilómetros que haya recorrido.

> Definí la clase `Colectivo` con sus métodos correspondientes. No te olvides que los colectivos son medios de transporte.

¡Dame una pista!

Solución | Biblioteca | Consola

```
1  class Colectivo < MedioDeTransporte
2    def recorrer!(kms)
3      @combustible -= kms*2
4    end
5
6    def maximo_personas
7      20
8    end
9
10 end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¿20 personas un colectivo? Yo he visto que lleve más. ¿Además sin pasajeros?

Bueno, bueno, es verdad. Vamos a mejorar un poco nuestro `Colectivo`.

Inconsciente colectivo

No hay 2 sin 3, tampoco hay 20 sin 21, o 30...

La verdad es que la cantidad de gente que puede entrar en un `Colectivo` es variable, y para simplificar las cosas vamos a decir que en un colectivo siempre entran personas.

Pero... ¿entonces no es un `MedioDeTransporte` ?

Sí, en realidad es un `MedioDeTransporte` , solo que responde distinto a `entran?` . Lo que podemos hacer es redefinir el método: si `Colectivo` define el método `entran?` va a evaluar ese código en lugar del de su superclase.

Ahora que sabemos que se pueden redefinir métodos, aprovechemos y cambiemos un poco más nuestra solución. Los colectivos siempre se inicializan con 100 de `@combustible` y con 0 `@pasajeros` .

> Redefiní los métodos `initialize` y `entran?` en la clase `Colectivo` .

Solución | Biblioteca | Consola

```
class Colectivo < MedioDeTransporte
  def initialize
    @combustible =100
    @pasajeros = 0
  end

  def entran?(cant)
    40>= cant
  end

  def recorrer!(kms)
    @combustible -= kms*2
  end

  def maximo_personas
    20
  end

end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¡Genial!

Esto de la herencia está buenísimo. Porque nos permite heredar el comportamiento de una superclase pero redefinir aquellas cosas que nuestras subclases hacen distinto. Pero cuidado, si tenemos que redefinir todo probablemente no necesitemos heredar en primer lugar.

¿Y qué pasa cuando en una subclase no hago lo mismo que en la superclase pero tampoco es taaaan distinto?

¡Vamos a verlo!

Es un trabajo para super

Bien sabemos que los colectivos también necesitan cargar combustible como cualquier `MedioDeTransporte`, pero ¡qué molesto para los pasajeros! Es por esto que cuando un `Colectivo` carga combustible, además de incrementarlo pierde a todos sus `@pasajeros`.

El tema es que si redefinimos `cargar_combustible!` en `Colectivo` vamos a repetir lógica con nuestra superclase `MedioDeTransporte`. No necesariamente, gracias al 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 SaludoDocente < Saludo
  def saludar
    super + " estudiantes"
  end
end
```

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

```
> mi_saludo = SaludoDocente.new
> mi_saludo.saludar
=> "Buen día estudiantes"
```

> ¡Ahora te toca a vos! Redefiní el método `cargar_combustible!` en `Colectivo`, de modo que haga lo mismo que cualquier `MedioDeTransporte` y además se quede sin pasajeros. Recordá utilizar `super` para evitar repetir lógica.

¡Dame una pista!

Solución | Biblioteca | Consola

```
1  class Colectivo < MedioDeTransporte
2    def initialize
3      @combustible =100
4      @pasajeros = 0
5    end
6
7    def entran?(cant)
8      40>= cant
9    end
10
11   def recorrer!(kms)
12     @combustible -= kms*2
13   end
14
15   def maximo_personas
16     20
17   end
18
19   def cargar_combustible!(combustible)
20     super
21     @pasajeros = 0
22   end
23
24 end
25
```

```
26
27
28
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

El regreso de los zombis

¿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?` son distintos, 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 definir únicamente en la superclase `Zombi`;
- **redefinir** en `SuperZombi` aquellos métodos cuya definición sea distinta a la de `Zombi`;
- definir **únicamente** en `SuperZombi` el comportamiento que es exclusivo a esa clase.

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

Solución | Consola

```
class Zombi
  def initialize(salud_inicial)
    @salud = salud_inicial
  end

  def salud
    @salud
  end

  def gritar
    "¡agrrrg!"
  end

  def sabe_correr?
    false
  end

  def sin_vida?
    @salud == 0
  end

  def recibir_danio!(puntos)
    @salud = [@salud - puntos * 2, 0].max
  end
end

class SuperZombi < Zombi
  def sabe_correr?
    true
  end

  def recibir_danio!(puntos)
    @salud = [@salud - puntos * 3, 0].max
  end

  def regenerarse!
    @salud = 100
  end
end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Prestá atención: lo que hicimos aquí es *parecido* a la herencia de los dispositivos, pero no igual. En nuestro ejemplo anterior, `Dispositivo` es una clase abstracta, porque nunca la vamos a instanciar, y nuestros dos dispositivos heredan de ella. Pero, ¿no queremos instanciar a la clase `Zombi` ?

Concretemos la herencia

A diferencia de lo que pasaba con la clase abstracta `Dispositivo` y sus subclases `Celular` y `Notebook`, `Zombi` es una clase concreta ¡y `SuperZombi` hereda de ella sin problemas!

¿Esto quiere decir que los zombis existen?

¡No, tranqui! Lo que quiere decir es que tiene sentido que existan instancias de la clase `Zombi`. Esto significa que podemos tener tanto objetos `SuperZombi` como `Zombi`.

En este caso, y al igual que con los dispositivos, las instancias de `SuperZombi` entenderán todos los mensajes que estén definidos en su clase, sumados a todos los que defina `Zombi`.

Y como ya aparecieron en muchos ejercicios, tanto los objetos de la clase `Zombi` como los de `SuperZombi` quieren `descansar!`. Cuando descansan una cantidad de minutos, su `@salud` se incrementa en esa cantidad.

> Definí el método `descansar!` en donde corresponda.

Solución | Consola

```
class Zombi
  def initialize(salud_inicial)
    @salud = salud_inicial
  end

  def salud
    @salud
  end

  def gritar
    "¡agrrrg!"
  end

  def sabe_correr?
    false
  end

  def sin_vida?
    @salud == 0
  end

  def recibir_danio!(puntos)
    @salud = [@salud - puntos * 2, 0].max
  end

  def descansar!(minutos)
    @salud += minutos
  end

end

class SuperZombi < Zombi
  def sabe_correr?
    true
  end

  def recibir_danio!(puntos)
    @salud = [@salud - puntos * 3, 0].max
  end

```

```
41   def regenerarse!
42     @salud = 100
43   end
44 end
45
46
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

La defensa

Zombis por aquí, super zombis por allá, ¿quién podrá ayudarnos?

¡Volvieron las clases `Sobreviviente` y `Aliado` ! Veamos parte de su comportamiento:

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

  def energia
    @energia
  end

  def beber!
    @energia *= 1.25
  end

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

class Aliado
  def initialize
    @energia = 500
  end

  def energia
    @energia
  end

  def beber!
    @energia *= 1.10
  end

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

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

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

> Último esfuerzo: definí una clase abstracta `Persona` que agrupe el comportamiento que se repite y hacé que las clases `Sobreviviente` y `Aliado` hereden de ella.

¡Dame una pista!

Solución | Consola

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

  def energia
    @energia
  end
```

```ruby

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

end


class Sobreviviente < Persona
  def beber!
    @energia *= 1.25
  end

end

class Aliado < Persona
  def initialize
    @energia = 500
  end

  def beber!
    @energia *= 1.10
  end

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

Enviar

¡Muy bien! Tu solución pasó todas las pruebas