



Zombi caminante

-

¡Te damos la bienvenida a la *invasión zombi*!

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

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

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

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

💡 ¡Dame una pista!

 Solución  Consola

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

```
19  end
20
21  end
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

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

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





Atacando un zombi

•

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

Además cuenta con un nivel de `energia`, que inicia en `1000`, pero todavía no haremos nada con él. Definí un método *getter* para este atributo.

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

💡 ¡Dame una pista!

 Solución  Consola

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

 Enviar

 ¡Muy bien! Tu solución pasó todas las pruebas

Ahora que [Juliana](#) sabe [atacar!](#) , veamos contra quién más se puede enfrentar...

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





Otro zombi caminante

•

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

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

[🔧 Solución](#)[>_ Consola](#)

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

```
28     @salud
29 end
30
31 def self.gritar
32     "¡agrrrg!"
33 end
34
35 def self.sabe_correr?
36     false
37 end
38
39 def self.recibir_danio!(puntos_de_danio)
40     @salud = [(@salud -= puntos_de_danio*2),0].max
41 end
42
43 end
44
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

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

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





¡¿Vivos?!

•

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

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

 Solución  Consola

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



```
30  @salud=100
31
32  def self.salud
33    @salud
34  end
35
36  def self.gritar
37    "¡agrrrrg!"
38  end
39
40  def self.sabe_correr?
41    false
42  end
43
44  def self.recibir_danio!(puntos_de_danio)
45    @salud = [(@salud -= puntos_de_danio*2),0].max
46  end
47
48  def self.sin_vida?
49    self.salud==0
50  end
51
52 end
53
```

 Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

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

Ahora ya es imposible no verlo: todo lo que se modifique en un zombie 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** zombies en lugar de dos? ¿Y si hubiese **cien**? ¡Cuántas veces habría que copiar y pegar!

Veamos una solución posible...

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





Clases

Si tenemos más de un objeto que se comporta **exactamente** de la misma forma, lo que podemos hacer es generalizar ese comportamiento definiendo 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
end
```

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

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

💡 ¡Dame una pista!

 Solución

 Biblioteca

 Consola

```
1 class Zombi
2
3   def initialize
4     @salud = 100
5   end
6
7   def salud
8     @salud
9   end
10
11  def gritar
12    "¡agrrrg!"
13  end
14
15  def sabe_correr?
16    false
17  end
18
19  def recibir_danio!(puntos_de_danio)
20    @salud = [(@salud -= puntos_de_danio*2),0].max
21  end
22
23  def sin_vida?
24    self.salud==0
25  end
26
27 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 zombies es el mismo, por eso pudimos generalizarlo.

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

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





Instancias

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

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

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

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

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

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

 Solución

 Biblioteca

 Consola

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

 Enviar

 ¡Muy bien! Tu solución pasó todas las pruebas

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

Hasta ahora estuvimos jugando con **objetos bien conocidos**, 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.

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





Al menos tenemos salud

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

7. Al menos tenemos salud

```
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`?

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

💡 ¡Dame una pista!

>_ Consola

</> Biblioteca

>



Esta guía fue desarrollada por Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)







Inicializando instancias

•

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

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

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

¡`anastasia` llega para combatir los zombis! Definí 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
3   def initialize
4     @energia = 1000
5   end
6
7   def energia
8     @energia
9   end
10
11  def atacar!(zombie, puntos_de_ataque)
12    zombie.recibir_danio!(puntos_de_ataque)
13  end
14
15 end
16
17 juliana = Sobreviviente.new
18 anastasia = Sobreviviente.new
19
```



 Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





Ahora sí: invasión

•

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

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

```
Vivero.agregar_planta! Planta.new  
Vivero.agregar_planta! Planta.new  
Vivero.agregar_planta! Planta.new
```



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

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



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

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

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

💡 ¡Dame una pista!

Solución Biblioteca Consola

```
1 class Sobreviviente  
2  
3   def initialize  
4     @energia = 1000  
5   end
```



```
6
7  def energia
8    @energia
9  end
10
11  def atacar!(zombie,puntos_de_ataque)
12    zombie.recibir_danio!(puntos_de_ataque)
13  end
14
15  def ataque_masivo!(caminantes)
16    caminantes.each{|caminante|atacar!(caminante,15)}
17  end
18
19 end
20
21 juliana = Sobreviviente.new
22 anastasia = Sobreviviente.new
23
24
25 caminantes = []
26 20.times {caminantes.push(Zombi.new)}
27
```



✓ ¡Muy bien! Tu solución pasó todas las pruebas

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

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

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

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





Al menos tenemos (menos) salud

•

`juliana` y `anastasia` estuvieron estudiando a los zombies 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
  def initialize(centimetros)
    @altura = centimetros
  end

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

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

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

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

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

 Solución Consola

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

 Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

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

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

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



Esta guía fue desarrollada por Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





Súper zombi

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

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

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

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

Solución

Biblioteca

Consola

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

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

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

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

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

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

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





Ejercitando

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

Pero también pueden `beber!` 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=@energia*1.25
21  end
22
23 end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

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

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





Aliados

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

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

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

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

Solución [Biblioteca](#) [Consola](#)

```
1 class Aliado
2   def initialize
3     @energia = 500
4   end
5
6   def energia
7     @energia
8   end
9
10  def atacar!(zombie, danio)
11    zombie.recibir_danio!(danio)
12    @energia=0.95*@energia
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=@energia*1.1
22  end
23
24 end
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Esta guía fue desarrollada por Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)



