Entrando en Calor

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

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

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

¡Dame una pista!

Solución | Consola

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

```
    TRUE
  end

end

module TimbaElLeon
  @dificultad = 25

  def self.violento?
    FALSE
  end

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

  def self.dificultad
    @dificultad
  end

end

module Metroide
  @nivel_espacial = 3


  def self.violento?
    if @nivel_espacial > 5
      TRUE
    end
  end

  def self.jugar!(un_tiempo)
    @nivel_espacial+=1
  end

  def self.dificultad
    100
  end

end

```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

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

Es importante que notes que todos estos objetos responden a los mismos mensajes: `dificultad`, `violento?` y `jugar!(un_tiempo)`. Como aprendiste con las golondrinas, 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!

Creando una lista

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

Para ello necesitamos definir un objeto, la `Juegoteca` , 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: definí un objeto `Juegoteca` que tenga un atributo `juegos` con su correspondiente getter. La `Juegoteca` tiene que tener en primer lugar el juego `CarlosDuty` , luego `TimbaElLeon` y por último `Metroide` .

¡Dame una pista!

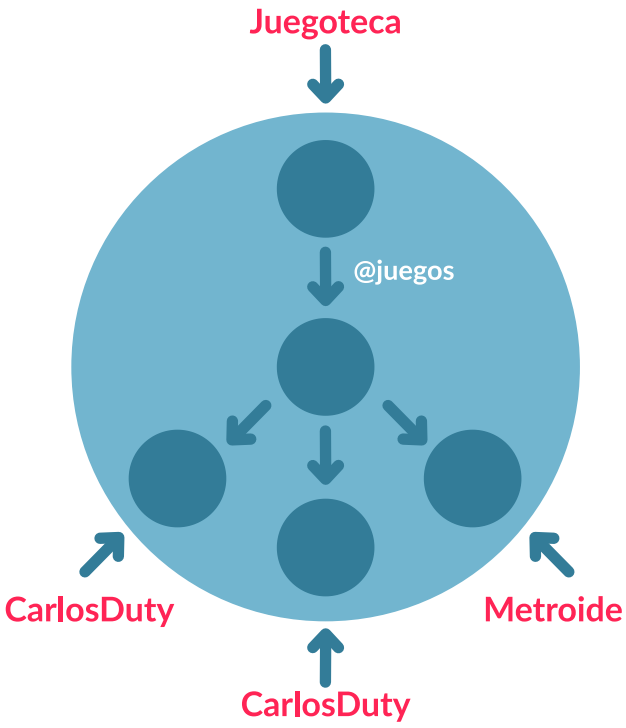
Solución | Biblioteca | Consola

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

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¡Excelente! Ya tenemos creada la `Juegoteca` con algunos juegos:



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

Algunos mensajes básicos

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

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

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

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

También podemos saber 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!

Consola | Biblioteca

>

Esta guía fue desarrollada por Felipe Calvo, Franco Bulgarelli, Mariana Matos, Gustavo Crespi bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0.

Información importante

Términos y Condiciones

Reglas del Espacio de Consultas

Set o no set

Hasta ahora sólo vimos un tipo de colección: las listas. ¡Pero hay más!

4. Set o no set

Otro tipo muy común de colecciones son los *sets* (*conjuntos*), los cuales tienen algunas diferencias con las listas:

- no admiten elementos repetidos;
- sus elementos no tienen un orden determinado.

Vamos a ver un ejemplo transforma una lista en un set utilizando `to_set` :

```
> numeros_aleatorios = [1,27,8,7,8,27,87,1]
> numeros_aleatorios
=> [1,27,8,7,8,27,87,1]
> numeros_aleatorios.to_set
=> #<Set: {1, 27, 8, 7, 87}>
```

Algo importante a tener en cuenta es que tanto las listas como los sets tienen mensajes en común. Dicho de otro modo, son polimórficos para algunos mensajes. Por ejemplo: `push` , `delete` , `include?` y `size` .

Sin embargo, los siguientes mensajes...

```
numeros_de_la_suerte = [6, 7, 42]
numeros_de_la_suerte.first
  # Nos retorna el primer elemento de la lista
numeros_de_la_suerte.last
  # Nos retorna el último de la lista
numeros_de_la_suerte.index 7
  # Nos retorna la posición de un elemento en la lista
```

... no podemos enviárselos a un set porque sus elementos no están ordenados.

Pero no te preocupes, todos lo que veamos de ahora en adelante en esta lección funciona tanto para listas como para sets.

Mejorando la Juegoteca

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

> Nuestra `Juegoteca` maneja `puntos` . Agregá el código necesario para que entienda los siguientes mensajes:
>
> - `puntos` : nos dice cuantos puntos tiene la `Juegoteca` . Inicialmente son 0.
> - `adquirir_juego!(un_juego)` : agrega el juego a la `Juegoteca` , y le suma 150 puntos.
> - `borrar_juego!(un_juego)` : quita un juego de la `Juegoteca` , pero no resta puntos.
> - `completa?` : se cumple si la `Juegoteca` tiene más de 1000 puntos y más de 5 juegos.
> - `juego_recomendable?(un_juego)` : es verdadero para `un_juego` si no está en la `Juegoteca` y es `violento?` .

¡Dame una pista!

Solución | Consola

```
1  module Juegoteca
2    @puntos = 0
3    @juegos=[CarlosDuty, TimbaElLeon, Metroide]
4
5    def self.juegos
6      @juegos
7    end
8
9    def self.puntos
10     @puntos
11   end
12
13   def self.adquirir_juego!(un_juego)
14     juegos.push un_juego
15     @puntos += 150
16   end
17
18   def self.borrar_juego!(un_juego)
19     juegos.delete un_juego
```

```
20    end
21
22    def self.completa?
23      @puntos > 1000 && juegos.size > 5
24    end
25
26    def self.juego_recomendable?(un_juego)
27      !(juegos.include?un_juego) && un_juego.violento?
28    end
29
30  end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

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

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

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

¿Bloques? ¿Eso se come?

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

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

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

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

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

¿Cuánto vale `un_numero` luego de las primeras dos líneas? Tené en cuenta que la secuencia de envío de mensajes en el *bloque* del primer ejemplo está **sin ejecutar**. En cambio, en el ejmplo de `otro_numero` estamos enviando el mensaje `call`. Por lo tanto:

- `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 argumentos para su aplicación. Por ejemplo, `sumar_a_otros_dos` recibe dos argumentos, 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 argumentos deseados al mensaje `call`:

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

Volvamos a los videojuegos...

En la *Biblioteca* podés ver el objeto `TimbaElLeon`. Para resolver este ejercicio, no nos interesa cómo están definidos los métodos de este objeto, solo queremos recordar los mensajes que entiende ¡por eso hay puntos suspensivos!

> 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.

¡Dame una pista!

Solución | Biblioteca | Consola

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

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

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

Filtrando quienes cumplen

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

8. Filtrando quienes cumplen

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

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

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

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

> *Mientras tanto, en nuestra juegoteca...*
>
> ¡Ahora te toca a vos! Definí el método `juegos_violentos` que retorna los juegos de la `Juegoteca` que cumplan `violento?`.

Solución | Biblioteca | Consola

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

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

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

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

Información importante

Términos y Condiciones

Reglas del Espacio de Consultas

El que busca encuentra

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


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

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

```
> uno_mayor_a_3
=> 4
```

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

> Veamos si se entiende: hacé que la `Juegoteca` entienda `juego_mas_dificil_que(una_dificultad)` , que retorna algún juego en la `Juegoteca` con más dificultad que la que se pasa como argumento.

Solución | Biblioteca | Consola

```
1  module Juegoteca
2    @juegos=[CarlosDuty, TimbaElLeon, Metroide]
3
4    def self.juegos
5      @juegos
6    end
7
8    def self.juego_mas_dificil_que(una_dificultad)
9      juego_dificil = juegos.find{|juego| juego.dificultad > una_dificultad}
10   end
11
12
13 end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

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

¿Alguno cumple? ¿Todos cumplen?

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

```
estudiantes.all? { |un_estudiante| un_estudiante.aprobo? }
```

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

```
estudiantes.any? { |un_estudiante| un_estudiante.aprobo? }
```

Definí los siguientes métodos en nuestra `Juegoteca`:

- `mucha_violencia?`: 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

```
module Juegoteca
  @juegos=[CarlosDuty, TimbaElLeon, Metroide]

  def self.juegos
    @juegos
  end

  def self.mucha_violencia?
    juegos.all? {|juego| juego.violento?}
  end

  def self.muy_dificil?
    juegos.any? {|juego| juego.dificultad > 25}
  end



end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

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

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

Información importante

Términos y Condiciones

Reglas del Espacio de Consultas

El viejo y querido map

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

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

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

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

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

Solución | Biblioteca | Consola

```
module Juegoteca
  @juegos=[CarlosDuty, TimbaElLeon, Metroide]

  def self.juegos
    @juegos
  end

  def self.juegos_violentos
    juegos_violent = juegos.select{|juego| juego.violento?}
  end

  def self.dificultad_violenta
    juegos_violentos.map {|juego| juego.dificultad}
  end

end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

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

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

4.0.

¿Cuántos cumplen? ¿Cuánto suman?

Volviendo a nuestra colección de estudiantes. Ya preguntamos si todo el grupo aprobó o si al menos alguien aprobó utilizando `all?` y `any?`. ¿Y si queremos saber **cuántos** aprobaron? Usamos `count`:

```
estudiantes.count { |un_estudiante| un_estudiante.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 la colección de estuidantes, por ejemplo, podemos hacer:

```
estudiantes.sum { |un_estudiante| un_estudiante.nota_en_examen }
```

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

¡Dame una pista!

Solución | Consola

```
module Juegoteca
  @juegos=[CarlosDuty, TimbaElLeon, Metroide]

  def self.juegos
    @juegos
  end

  def self.juegos_violentos
    juegos_violent = juegos.select{|juego| juego.violento?}
  end

  def self.promedio_de_violencia
     juegos_violentos.sum{|juego| juego.dificultad}/juegos_violentos.count{|juego| juego.violento?}
  end

end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Jugando a todo

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

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

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

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

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

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

Solución | Biblioteca | Consola

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

Enviar

¡Muy bien! Tu solución pasó todas las pruebas