En un cumpleaños cada persona recibe una bolsita con golosinas y como no sabe cuál le gusta más quiere probarlas a todas 🍭. Sabemos que cuando una persona:

- prueba un `PaqueteDeGomitas`, el paquete pierde una unidad;
- prueba un `Alfajor`, el peso de esta golosina disminuye en 10 gramos su peso;
- prueba un `Chupetin`, no le pasa nada a la golosina.

> Definí el método `probar_golosinas!` en la clase `Persona` y el método `degustar!` en los distintos tipos de golosinas.

Solución | Consola

```ruby
class Persona
  def initialize(unas_golosinas)
    @golosinas = unas_golosinas
  end

  def probar_golosinas!
    @golosinas.each {|golosina|golosina.degustar!}
  end
end



class PaqueteDeGomitas
  def initialize(cant_unidades)
    @unidades = cant_unidades
  end


  def degustar!
    @unidades -= 1
  end

end

class Alfajor
  def initialize(un_peso)
    @peso = un_peso
  end
  def degustar!
    @peso -= 10

  end

end

class Chupetin
  def degustar!
    @degustar
  end
end
```

▶ Enviar

# Ejercicio 8: Ejercicio 8

En una empresa de mensajería llamada `MensajeriaDelNorte` tienen un conjunto con todos los paquetes 📦. Al cerrar el día es necesario saber cuales quedaron pendientes de entrega para el día siguiente. Teniendo en cuenta que cada paquete sabe responder al mensaje `falta_entregar?`...

> Definí en Ruby el método `cuantos_son_pendientes` que responda a cuántos paquetes están pendientes de entrega en `MensajeriaDelNorte`.

Solución | Consola

```ruby
module MensajeriaDelNorte
  @paquetes = [Paquete1, Paquete2, Paquete3, Paquete4, Paquete5, Paquete6]

  def self.falta_entregar?
    true
  end

  def self.cuantos_son_pendientes
      @paquetes.count {
      |paquete| paquete.falta_entregar?
      }
  end

end
```

▶ Enviar

✅ ¡Muy bien! Tu solución pasó todas las pruebas

# Ejercicio 7: Ejercicio 7

¡Dejemos atrás a JavaScript para pasar a Ruby!

Vamos a desarrollar parte de un juego en el cual es necesario recolectar diversos recursos. En este caso debemos modelar al personaje `Tidus` que se encarga de juntar arcilla para construcción. Sabemos que `Tidus`:

- inicialmente tiene 200 de arcilla;
- puede conseguir arcilla de a 100 por vez;
- si tiene más de 1500 de arcilla diremos que `puede_construir_casa?`.

> Definí en Ruby, el objeto `Tidus` que tenga un atributo `@arcilla` con su getter. El objeto entiende los mensajes `recolectar!` (que aumenta en 100 su cantidad de arcilla) y `puede_construir_casa?`. No te olvides de inicializar el atributo `@arcilla` con el valor correspondiente.

Solución | Consola

```ruby
module Tidus
  @arcilla = 200
  def self.arcilla
    @arcilla
  end

  def self.recolectar!
    @arcilla += 100
  end

  def self.puede_construir_casa?
    @arcilla > 1500
  end
end
```

▶ Enviar

✅ ¡Muy bien! Tu solución pasó todas las pruebas

# Ejercicio 6: Ejercicio 6

En un comercio de indumentaria guardan registro de los productos que venden junto con su precio y su descuento a aplicar:

```
let unaRemera = {
  producto: "Remera XL negra",
  precioBase: 700,
  descuento: 200
}

let unaBermuda = {
  producto: "Bermuda M rosa",
  precioBase: 1000,
  descuento: 100
}
```

> Definí la función `resumenDeInformacion` que permita obtener un resumen de la información registrada. Por ejemplo:

```
resumenDeInformacion(unaRemera)
"El artículo Remera XL negra se vende a $500"

resumenDeInformacion(unaBermuda)
"El artículo Bermuna M rosa se vende a $900"
```

Solución | Consola

```
function resumenDeInformacion (ropa) {
  return "El artículo " + ropa.producto + " se vende a $" +
(ropa.precioBase - ropa.descuento).toString ()
}
```

▶ Enviar

✅ ¡Muy bien! Tu solución pasó todas las pruebas

# Ejercicio 5: Ejercicio 5

Siempre que llovió paró, y lo mejor de eso es cuando sale el sol y aparece un arcoíris. Los arcoíris tienen siete colores y queremos filtrar a partir de una lista cuáles son parte de ellos. Para eso tenemos la función `perteneceAlArcoiris` que nos dice si un color está entre esos siete:

```
perteneceAlArcoiris('azul')
true

perteneceAlArcoiris('naranja')
true

perteneceAlArcoiris('negro')
false
```

> Definí la función `obtenerColores` que a partir de una lista de colores nos retorne una lista con aquellos colores que sean parte de un arcoíris. Por ejemplo:
>
> ```
> obtenerColores(['verde','violeta','negro','salmón','rojo'])
> ['verde','violeta','rojo']
> ```

Solución | Consola

```
function obtenerColores (lista) {
  let colores = [];
  for (let color of lista) {
    if (perteneceAlArcoiris (color)) {
      agregar (colores, color)
    }

  }

  return colores;
}
```

▶ Enviar

✅ ¡Muy bien! Tu solución pasó todas las pruebas

# Ejercicio 4: Ejercicio 4

Una librería que vende libros e historietas nos pidió una función para generar los identificadores de sus productos. En el caso de los libros el mismo comienza con 14 y en el caso de las historietas con 44. En ambos casos al final llevan un guión y un número extra que recibiremos como argumento:

```
armarIdentificador("libro", 48)
"14-48"
armarIdentificador("historieta", 19)
"44-19"
```

Definí la función `armarIdentificador` que genere el identificador de un producto a partir de un string clasificatorio y el número que se coloca al final.

Solución | Consola

```
function armarIdentificador (producto, numero) {
  if (producto === "libro") {
    return "14" + "-" + numero.toString()
  } else {
     return "44" + "-" + numero.toString ()
    }

}
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

# Ejercicio 3: Ejercicio 3

Ari está desarrollando juegos de mesa para jugar en línea. Al hacerlo notó que la mayoría requiere del uso de un dado por lo que nos solicitó que desarrollemos una función que a partir de un número nos diga si es un valor válido, es decir, si está entre 1 y 6:

```
esUnNumeroDeDado(1)
true

esUnNumeroDeDado(20)
false
```

Definí la función `esUnNumeroDeDado` que recibe un número y nos dice si es un valor válido de un dado.

Solución | Consola

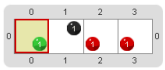
```
function esUnNumeroDeDado (numero) {
  return 1 < numero <= 6 && numero > 0 && numero < 7;
}
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

# Ejercicio 2: Ejercicio 2

Una fábrica de chocolates nos pidió un procedimiento que se encargue de armar una caja de bombones con distintos sabores. Actualmente venden bombones de frutilla, menta y chocolate amargo que representaremos con bolitas de color `Rojo`, `Verde` y `Negro` respectivamente. Las cajas tienen cuatro bombones que recibiremos como argumento y arman la caja con los mismos, uno al lado del otro. Por ejemplo, si lo invocamos haciendo `AgregarBombones(Verde, Negro, Rojo, Rojo)` la caja deberá verse así:



Definí el procedimiento `AgregarBombones` que recibe 4 colores y arma la caja de bombones. El cabezal comienza en el extremo Sur Oeste y no importa dónde finaliza.

```
procedure AgregarBombones (colorA, colorB, colorC, colorD) {
  Poner (colorA)
  Mover(Este)
  Poner(colorB)
  Mover (Este)
  Poner(colorC)
  Mover (Este)
  Poner(colorD)
}
```

▶ Enviar

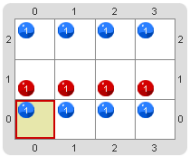
¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:



# Ejercicio 1: Ejercicio 1

Un grupo de estudiantes nos solicitó que hagamos el modelo de la bandera de su curso utilizando Gobstones . Esta bandera tiene 3 franjas horizontales de 4 celdas cada una y se ve de la siguiente forma:



Es decir, las franjas de los extremos de color `Azul` y la del medio de color `Rojo`.

Creá el programa que replique la bandera del curso. El cabezal comienza en el extremo Sur Oeste y no importa dónde finaliza.

```
procedure LineaAzul () {
  repeat (3){
  Poner (Azul)
  Mover (Este)
    }
  Poner (Azul)
}

procedure LineaRojo () {
  repeat (3){
  Poner (Rojo)
  Mover (Este)
    }
  Poner (Rojo)
}

program {
  LineaAzul ()
  Mover (Norte)
  IrAlBorde (Oeste)
  LineaRojo ()
  Mover (Norte)
  IrAlBorde (Oeste)
  LineaAzul ()

}
```

▶ Enviar

¡Muy bien! Tu solución pasó todas las pruebas