Variables

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 saludo_formal = "buen día"
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¡Momento, momento!

¿Qué sucedió aquí? Hasta ahora habíamos visto que tenemos objetos y mensajes, y sólo le podíamos enviar mensajes a los objetos, como `Pepita` , `14` , u `"hola"` . ¿Le acabamos de enviar un mensaje a una variable?

Sí y no. Veamos por qué...

Las variables son referencias

Hasta ahora venimos insistiendo con que, en la programación en objetos, le enviamos mensajes a los objetos. ¡Y no mentimos!

2. Las variables son referencias

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 enviale el mensaje `size()`.

¡Dame una pista!

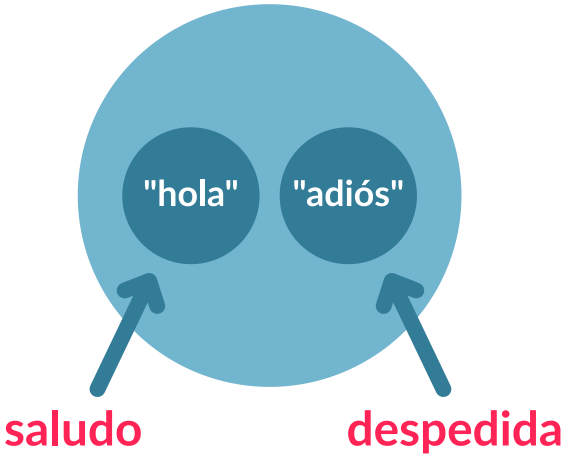
Solución | Consola

```
despedida = "adiós"
despedida.size()
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¡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!

Referencias implícitas

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

3. Referencias implícitas

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?:

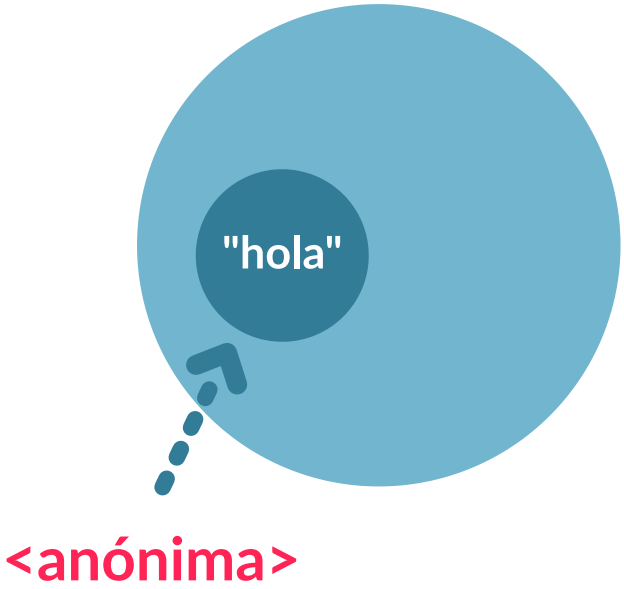
```
#¿A qué referencia el envío upcase?
"ni hao".upcase

#¿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"
```



. 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`

```
>
```

Múltiples referencias

Supongamos que tenemos el siguiente programa:

4. Múltiples referencias

```
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?

> ¡Averigualo! **Declará las variables** `otro_saludo` y `despedida` como en el ejemplo de más arriba, y realizá las siguientes consultas utilizando `equal?`:
>
> - `> "buen día".equal? "buen día"`
> - `> despedida.equal? "buen día"`
> - `> otro_saludo.equal? otro_saludo`
> - `> despedida.equal? otro_saludo`
>
> ¡Ahora sacá tus conclusiones viendo que responde en cada caso!

```
>
```

Identidad, revisada

El mensaje `equal?` nos dice si dos objetos son el mismo. Veamos qué pasó con las pruebas del ejercicio anterior:

5. Identidad, revisada

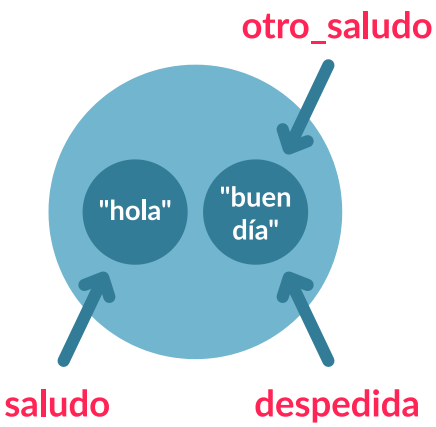
```
otro_saludo = "buen día" # se crea la variable otro_saludo que referencia al objeto "buen día"
despedida = otro_saludo # se crea la variable despedida que, por asignarle la referencia otro_saludo, apunta al mismo objeto
```

```
> "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
```

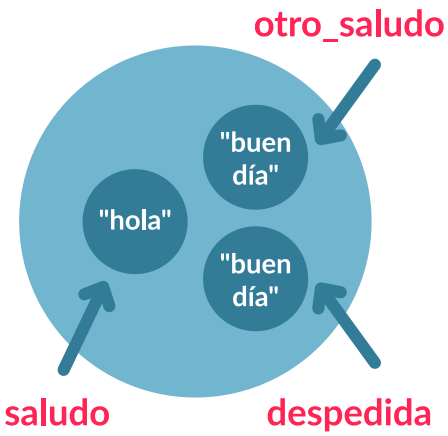
¿Por qué? ¡Simple! Ambas referencias, `otro_saludo` y `despedida`, apuntan al mismo objeto. La moraleja es que declarar una variable significa agregar una nueva referencia al objeto existente, en lugar de copiarlo:



Distinto sería si hacemos:

```
otro_saludo = "buen día"
despedida = "buen día"
```

Lo cual da como resultado este ambiente:



Veamos otro ejemplo. Si tuvieramos el siguiente código...

```
persona = "Graciela"
hija_de_hector = "Graciela"
hermana_de_tito = persona
hija_de_elena = "Gracielita"
hermana_de_ana = hermana_de_tito
mama_de_gustavo = "hermana_de_ana"
tia_de_gonzalo = hija_de_hector
```

... podríamos decir que solo `hermana_de_tito` y `hermana_de_ana` referencian al mismo objeto que `persona`.

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?

Equivalencia

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*.

Por ejemplo, en este caso...

```
procer = "San Martín"
avenida = "San Martín"
ciudad = "San Martín"
```

... las 3 referencias **distintas** apuntan a objetos equivalentes entre sí, pero no idénticos.

¡Cuidado! 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 la implementan:
  - Listas
  - Números
  - Strings
  - Booleanos

Información importante

Términos y Condiciones

Reglas del Espacio de Consultas

Objetos bien conocidos

¿Y qué hay de los objetos que veníamos definiendo hasta ahora? Por ejemplo a `Fito`, le aumenta la felicidad cuando come:

7. Objetos bien conocidos

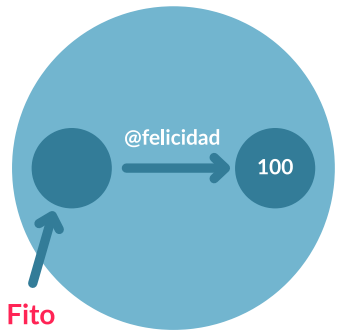
```
module Fito
  @felicidad = 100

  def self.comer!(calorias)
     @felicidad += calorias * 0.001
  end

  def self.felicidad
    @felicidad
  end
end
```

A objetos como `Fito` se los conocen como *objetos bien conocidos*: cuando los definimos no sólo describimos su comportamiento (`comer!(calorias)` 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. Definí un objeto `AbuelaClotilde` que entienda un mensaje `alimentar_nieto!`, que haga `comer!` 2 veces a `Fito`: primero con 2000 calorias, y luego con 1000 calorías; ¡el postre no podía faltar!.

Solución | Consola

```
1 module AbuelaClotilde
2
3   def self.alimentar_nieto!
4      Fito.comer!(2000)
5      Fito.comer!(1000)
6   end
7
8 end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

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 entiendas que *siempre* estamos enviando el mensaje al objeto a través de una referencia.

Atributos y parámetros

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

8. Atributos

Por ejemplo, si la golondrina `Pepita` conoce siempre su ciudad actual...

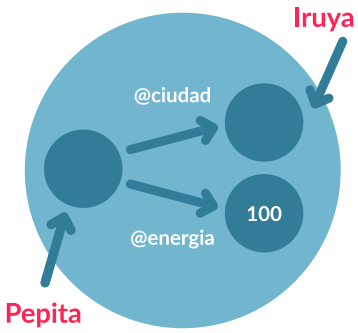
```
module Pepita
  @energia = 100

  def self.volar_en_circulos!
    @energia -= 10
  end

  def self.ciudad=(una_ciudad)
      @ciudad = una_ciudad
  end

  def self.ciudad
    @ciudad
  end
end
```

Y en algún momento esta pasa a ser `Iruya`, el diagrama de objetos será el siguiente:



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`.

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

Solución | Biblioteca | Consola

```
1 Pepita.ciudad=Iruya
```

Enviar

¡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.

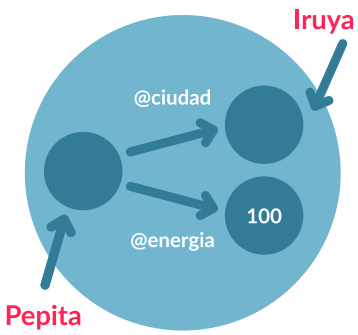
Lo 100to

Miremos este método con más detenimiento:

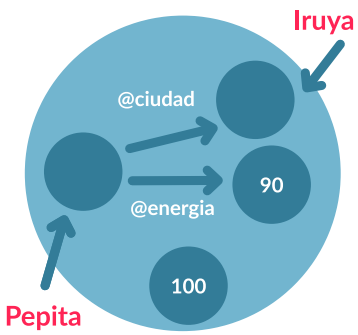
```
def self.volar_en_circulos!
   @energia = @energia - 10
end
```

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. `@energia` es una referencia a un objeto, que inicialmente *apunta* al objeto `100` :



Luego, la operación de asignación cambia ese apuntador, que pasa a referenciar al `90` :



¡Veamos si se entiende!

En este código...

```
module Pepita
  @energia = 100

  def self.volar_en_circulos!
    @energia -= 10
  end

  def self.ciudad=(una_ciudad)
    @ciudad = una_ciudad
  end
end

module Iruya
end
```

...si bien:

- `Pepita` e `Iruya` son objetos bien conocidos;

- `@energia` y `@ciudad` son atributos;
- y `una_ciudad` es un parámetro;

¡Todas son referencias!

Objetos compartidos

¿Te acordás de Fito? `Fito` también tiene un amigo, `Juli`. `Juli` es nieto de `AbueloGervasio`. Cuando `Juli` es feliz `Fito` es feliz:

```
module Fito
  def self.amigo=(un_amigo)
    @amigo = un_amigo
  end

  def self.es_feliz_como_su_amigo?
    @amigo.felicidad > 105
  end
end
```

> Creá un programa que inicialice al `amigo` de `Fito` y al `nieto` de `AbueloGervasio` de forma que ambos conozcan al mismo objeto (`Juli`).
> Luego, hacé que el **abuelo** alimente a su nieto 3 veces. ¿Qué pasará con `Fito` ? ¿Se pondrá feliz?

¡Dame una pista!

Solución | Consola

```
Fito.amigo=Juli
AbueloGervasio.nieto=Juli

3.times {AbueloGervasio.alimentar_nieto!}
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

En el programa que acabás de crear, que probablemente se vea parecido a esto...

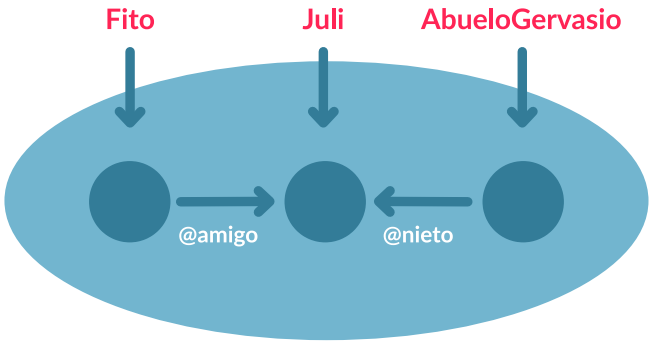
```
Fito.amigo = Juli
AbueloGervasio.nieto = Juli

#Si antes de alimentar al nieto preguntáramos Fito.es_feliz_como_su_amigo?, respondería false

3.times { AbueloGervasio.alimentar_nieto! }
```

... `Juli` es un **objeto compartido**: tanto el abuelo como su amigo lo conocen. La consecuencia de esto es que cuando su abuelo le da de comer le aumenta la felicidad, y su amigo 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.



Para cerrar

Antes de terminar nos topamos con un último problema: `Jor` se encarga de la cocina en un restaurante de pastas y le encanta `picantear!` su plato del día. Sin embargo, `Luchi`, su ayudante, no comparte ese gusto y por suerte está ahí para `suavizar!` las distintas comidas del establecimiento.

- Cada vez que `Jor` picantea el plato del día le agrega 5 ajíes.
- ¡Hoy hay `Fideos`! Decimos que los `Fideos` están picantes si el plato tiene más de 2 ajíes. Inicialmente no tiene ajíes.
- Cuando descartamos la salsa de los `Fideos` dejan de tener ajíes.
- `Luchi` necesita saber qué plato suavizar y cuántos ajíes sacarle para hacerlo. Si el plato tiene más de 10 ajíes, directamente descarta la salsa para poder usarla en otra comida.

Definí los objetos necesarios para que podamos hacer lo siguiente:

```
#Configurar el plato del día de Jor
Jor.plato_del_dia = Fideos
#Que Jor le agregue picante
Jor.picantear!
#Preguntar si está picante
Fideos.picantes?
#Que Luchi suavice el plato que reciba como argumento
Luchi.suavizar! Fideos, 3
#Descartar la salsa
Fideos.descartar_la_salsa!
```

¡Dame una pista!

Solución | Consola

```
module Fideos
  @ajies = 0

  def self.ajies
    @ajies
  end

  def self.agregar_ajies!(cantidad)
      @ajies+=cantidad
  end

  def self.quitar_ajies!(cantidad)
    @ajies-=cantidad
  end

  def self.descartar_la_salsa!
    @ajies=0
  end

  def self.picantes?
    @ajies>2
  end

end

module Jor

  def self.plato_del_dia
    @platoDelDia
  end

```

```
  def self.plato_del_dia=(plato)
    @platoDelDia=plato
  end

  def self.picantear!
    @platoDelDia.agregar_ajies!(5)
  end

end

module Luchi

  def self.suavizar! (plato_del_dia,quitar)
    if plato_del_dia.ajies>10
      plato_del_dia.descartar_la_salsa!
    else
      plato_del_dia.quitar_ajies!(quitar)
    end
  end

end

```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas