Retomando el concepto de Clase

- Todos los objetos son instancia de una clase
- La clase es un objeto
- Los métodos están en las clases
- Las definiciones de los atributos que va a tener una instancia están en su clase (o sea, los nombres de sus variables de instancia)

Si volvemos al ejemplo de pepita pero visto con clases

Object subclass: #Golondrina
  instanceVariableNames: 'energia ubicacion'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ejemplo'

```
Golondrina >> come: gramos
    energia := energia + (gramos * 4)

Golondrina >> vola: kilometros
    energia := energia + (self energiaNecesariaParaVolar: kilometros)

Golondrina >> energiaNecesariaParaVolar: kms
    ^kms + 10

Golondrina >> initialize
    energia := 0

Golondrina >> ubicacion: unLugar
    ubicacion := unLugar

Golondrina >> energia
    ^energia

Golondrina >> andaA: unLugar
    | kilometrosARecorrer |
    kilometrosARecorrer := unLugar distanciaHasta: ubicacion.
    (self tieneEnergiaParaVolarA: unLugar)
        ifTrue:
       [ self vola: kilometrosARecorrer.
              ubicacion := unLugar ]

Golondrina >> tieneEnergiaParaVolar: kilometros
    ^(self energiaNecesariaParaVolar: kilometros) <= energia
```

Object subclass: #Ciudad
  instanceVariableNames: 'kmEnRuta'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ejemplo'

```
Ciudad >> distanciaHasta: otroLugar
```

```
    ^(self kmEnRuta - otroLugar kmEnRuta) abs

Ciudad >> kmEnRuta
    ^kmEnRuta

Ciudad >> kmEnRuta: unNumero
    kmEnRuta := unNumero
```

Bloques

¿Qué es un bloque?

Un bloque es un objeto, y por lo tanto podemos hacer con él lo mismo que hacíamos con los demás objetos esto es:

- Enviarle mensajes
- Pasarlo como parámetro de algún mensaje
- Hacer que una variable lo referencie

Un objeto bloque representa un cacho de código que no se ejecutó, ese código solo se ejecutará si alguien le manda un mensaje que le indique hacerlo.

Si escribo en un workspace

```
x := 0.

[ x := x + 1].
```

¿A qué objeto apunta la variable x después de ejecutar esas 2 líneas de código? Claro, al objeto cero (`0`)

Para que la variable x apunte al objeto uno (`1`) tenemos que decirle al bloque que se ejecute.

```
x := 0.

[ x := x + 1] value.
```

Ahora sí, después de ejecutar esas 2 líneas de código la variable x va a apuntar al objeto uno (`1`).
Es importante darse cuenta que `value` es un mensaje que le llega al objeto bloque `[ x := x + 1].`

El mensaje `value` hace que se ejecute el código que está dentro del bloque y que se retorne el último objeto devuelto por la última sentencia del bloque.

Ejemplo:

```
[ pepita come: 14.
  5 factorial.
  pepe sueldo.
  3 between: 1 and: 5. ] value.
```
"Esto devuelve el objeto `true` porque es lo que devuelve el último envío de mensaje dentro del bloque"

Bloques como funciones

También se puede ver a los bloques como objetos que representan una función sin nombre (o sea, una función anónima).

El bloque `[ 1 ]` es una función constante que siempre devuelve 1 si le decís que se ejecute.

`[ 1 ] value.` "Esto devuelve el objeto uno `(1)`"

Pero el chiste de las funciones es que reciban parámetros y los bloques también pueden recibir parámetros

```
[ :parametro1 :parametro2 :parametro3 ... :parametroN | cuerpoDelBloque ]
```

Ejemplos:

```
f(x) = 2x
[ :x | 2 * x ] value

g(x) = x^2
[ :x | x raisedTo: 2 ]

h(x,y) = x^2 + y^2
[ :x :y | (x raisedTo: 2) + (y raisedTo: 2) ]

s(a,b,c) = cos a + sen b + c
[ :a :b :c | a cos + b sin + c ]
```

¿Cómo le hacemos para usar bloques que tienen parámetros?

```
f(4)
[ :x | 2 * x ] value: 4
```
"Esto devuelve el objeto `8`"

```
h(4,3)
[ :x :y | (x raisedTo: 2) + (y raisedTo: 2) ] value: 4 value: 3
```
"Esto devuelve el objeto `25`"

```
s(5,2,1)
[ :a :b :c | a cos + b sin + c ] value: 5 value: 2 value: 1
```
"Esto devuelve el objeto `2.192959612288908`"

Jugando con los bloques

```
bloque1 := [ :unAve | unAve vola: 20. unAve come: 30 ].
bloque2 := [ :unAve | unAve vola: 50. unAve come: 14 ].
```

`bloque1 value: pepita.` *"Esto hace que pepita vuele 20 kilometros y morfe 30 gramos de alpiste"*
`bloque2 value: pepita.` *"Esto hace que pepita vuele 50 kilometros y morfe 14 gramos de alpiste"*

`bloque1 value: pepona`. *"Esto hace que pepona vuele 20 kilometros y morfe 30 gramos de alpiste"*

¿Cómo funciona el #ifTrue: y el #ifFalse:?

Si en un workspace escribimos

```
(pepita energia > 0) ifTrue: [ pepita come: 30 ]
```

Pensando en términos de objeto y mensaje (mensaje = selector + parámetros) qué está pasando acá???

El objeto receptor del mensaje **`ifTrue:`** es el objeto que me devuelve `pepita energia > 0`, ese objeto puede ser `true` o `false`.

Si el receptor es `true` queremos que el bloque `[ pepita come: 30 ]` se ejecute.
Si el receptor es `false` NO queremos que el bloque `[ pepita come: 30 ]` se ejecute.

Entonces el objeto que tiene la responsabilidad de saber si el bloque debe o no ejecutarse es el booleano receptor del mensaje **`ifTrue:`**.

Siendo `false` la única instancia de la clase `False` y `true` la única instancia de la clase `True`, la implementación del método **`ifTrue:`** en cada una de las clases es

```
True >> ifTrue: unBloque
    "self apunta a true entonces queremos que se ejecute el bloque"
    ^unBloque value

False >> ifTrue: unBloque
    "self apunta a false entonces NO queremos que se ejecute el bloque"
    ^nil
```

Colecciones

Pensemos en modelar un tren usando objetos. Un tren tiene uno o varios vagones.

Los objetos vagones saben decirnos:

- Cuál es la máxima cantidad de pasajeros que pueden llevar
- Si se considera un vagón liviano o no
- Si ofrece servicio de bar

Tenemos que:

1. Armar un tren con uno o varios vagones
2. Saber cuántos vagones livianos tiene un tren
3. Saber cuál es la máxima cantidad de pasajeros que puede transportar el tren
4. Saber si un tren es liviano (si todos sus vagones son livianos)
5. Saber si un tren es VIP (si alguno de sus vagones ofrece servicio de bar)

Pensando en un workspace qué mensajes enviaría y a quién se los enviaría para resolver estos requerimientos, podría llegar a algo como lo siguiente:

```
trenDeLaAlegria agregarVagon: vagonDeBarney.
trenDeLaAlegria cantidadVagonesLivianos.
trenDeLaAlegria maximaCantidadDePasajeros.
...
```

Todos los requerimientos que tenemos que resolver van a terminar resolviéndose en el comportamiento de un objeto tren. Así como tengo al trenDeLaAlegria, quiero tener a muchos otros trenes iguales con el mismo comportamiento (que entiendan los mismos mensajes). Y... ¿de dónde los saco?

Recordemos el concepto de clase. Una clase es el "molde" de mis objetos, en donde se define su comportamiento (a través de los métodos) y qué atributos van a tener (a través de las variables definidas en ella). Sabiendo que todo objeto es instancia de una clase, como obtenemos un nuevo tren en nuestro ejemplo?

```
trenDeLaAlegria:= Tren new.
```

Aparece la clase Tren, en donde vamos a definir el comportamiento que van a tener los trenes. Enviándole el mensaje new a la clase Tren obtengo un nuevo objeto (una nueva instancia).

Empecemos a pensar cómo puedo "armar" un tren. A un tren le voy a querer agregar vagones de esta manera:

```
trenDeLaAlegria agregarVagon: vagonDeBarney.
```

Pensando en cómo será el método correspondiente al mensaje `#agregarVagon:`, me doy cuenta de que voy a necesitar que el tren guarde una referencia al vagón que le quiero agregar.
Si le agrego otro vagón más, necesito que referencie a 2 vagones en vez de uno. ¿Y si agrego *otro* vagón más?
Tengo que pensar que la cantidad de vagones varía de tren en tren, entonces ¿cuántas referencias (variables de instancia) que apunten a un vagón necesita tener un tren?

Supongamos que tomamos una decisión no muy coqueta y hacemos que un tren tenga 10 referencias a vagones (vagon1, vagon2, vagon3, ..., vagon10). Al armar un tren me puedo encontrar con las siguientes situaciones:

- Necesito agregarle 5 vagones, por lo tanto 5 de esas 10 variables no las uso y quedan referenciando a nil. Es incómodo (porque tendría que estar distinguiendo entre las variables que apuntan a un vagón y las que apuntan a nil), pero me sirve.
- Necesito agregarle 15 vagones... me quedé corto de referencias. ¿Tengo que agregarle más referencias mientras armo el objeto vagón? Eso no suena muy feliz.
- Necesito agregarle 10 vagones y encaja justito, pero aunque el vagón me haya quedado bien armado, cuando tenga que trabajar sobre todos los vagones va a ser incómodo mirar variable por variable para hacer lo mismo con cada una.

Esto nos lleva a que tener una referencia por cada vagón no es el camino apropiado. Necesito conocer vagones pero la cantidad de vagones que conozco es desconocido; lo que necesito es poder tratarl a los vagones como **un solo conjunto**. A estos conjuntos de objetos con cantidad de elementos variable los vamos a llamar **colecciones,** y por supuesto, las colecciones también son objetos.

En otras palabras: **una colección es un objeto que representa un conjunto de objetos**.

¿Y qué puedo hacer con una colección?

- Agregarle un elemento
- Sacarle un elemento
- Preguntarle cuántos elementos tiene
- Preguntarle si está vacío
- Saber cuáles de sus elementos cumplen una cierta condición
- Saber si todos los elementos de una colección cumplen una cierta condición
- Etc.

Entonces, empecemos con el primer requerimiento. Necesito agregarle un vagón a un tren, entonces, a un tren le pido que agregue un vagón:

```
Tren>>agregarVagon: unVagon
     vagones add: unVagon
```

Las colecciones entienden el mensaje `#add:` para agregar un elemento en ellas. Para que esto funcione, la variable vagones tiene que estar apuntando a una colección. ¿Cuándo defino eso? Podemos hacerlo al inicializar el objeto:

```
Tren>>initialize
     vagones:= Set new.
```

En este código vemos que vagones va a apuntar a una nueva instancia de la clase **Set**. Un objeto Set representa un conjunto y es uno de los sabores de colecciones que vamos a ver durante la cursada. Un conjunto no tiene orden entre sus elementos y tampoco tiene elementos repetidos.

Listo, ya puedo armar un tren con la cantidad de vagones que quiera, y voy a poder tratar a estos vagones como un conjunto.

Ahora queremos saber cuántos vagones livianos tiene un tren. Para eso, en principio debería poder saber cuáles son los vagones livianos de un tren para luego saber cuántos son.

Desde un workspace le puedo pedir a un tren su conjunto de vagones livianos

`trenDeLaAlegria vagonesLivianos.` "Esto me devuelve un conjunto que solo tiene a los vagones livianos del tren"

Ahora, para saber la cantidad de vagones livianos lo único que tengo que hacer es preguntarle a ese conjunto cuantos elementos tiene, eso se hace con el mensaje `#size`.

```
Tren>>cantidadVagonesLivianos
  ^self vagonesLivianos size
```

Pero bien, nos queda hacer el método vagonesLivianos. Para saber cuáles son los vagones livianos de un tren, debería "filtrar" de la colección de vagones a los que son livianos, o dicho de otra manera, seleccionar de la colección sólo a los que son livianos. Eso lo hacemos así:

```
Tren>>vagonesLivianos
     ^vagones select:[:vagon | vagon esLiviano]
```

El mensaje que le estamos enviando a la colección es `#select:` con un bloque de un parámetro como argumento. Este bloque representa a la condición que queremos que se verifique para los elementos que queremos seleccionar de la colección, y se evaluará con cada elemento de la colección que recibe el mensaje `#select:` (por eso espera un parámetro).
¿Y qué me devuelve `#select:`? Me devuelve una nueva colección que contiene sólo a los elementos que cumplieron con la condición que le llegó al select:, es decir, los elementos con los que la evaluación del bloque devolvió **true**.

Podemos ver un ejemplo con una colección de números:

```
variosNumeros := Set new.
variosNumeros add: 1.
variosNumeros add: 10.
variosNumeros add: 15.
variosNumeros add: 6.
variosNumeros add: 42.
```

`variosNumeros select:[:numero | numero even]` ---> me devuelve una nueva colección que tiene sólo los números pares que hay en la colección variosNumeros, es decir, una colección con los elementos 10, 6 y 42.

`variosNumeros size` ---> me devuelve 5 porque la colección variosNúmeros no se ve afectada por el select:

Para saber la cantidad máxima de pasajeros que puede llevar el tren, necesitamos sumar la cantidad máxima de pasajeros que puede llevar cada uno de sus vagones. Una manera de hacer esto es obtener el conjunto de cantidades máximas de pasajeros de todos los vagones, y luego sumarlas.

```
Tren>>cantidadMaximaDePasajeros
    ^(vagones collect:[:vagon | vagon cantidadMaximaPasajeros]) sum
```

Para obtener el conjunto de cantidades máximas de pasajeros de cada vagón del tren, le enviamos el mensaje **`#collect:`** a la colección, que recibe un bloque de un parámetro y devuelve otra colección donde cada elemento es el resultado de haber evaluado el bloque sobre un elemento de la colección receptora del mensaje.

Viendo esto en un ejemplito rápido sobre la misma colección de variosNumeros ya mencionada, podemos evaluar esto:

```
variosNumeros collect:[:numero | numero * 2]
```

En este caso, al enviarle el mensaje #collect: con ese bloque de parámetro a la colección variosNumeros, voy a obtener otra colección con el doble de cada uno de los números de variosNumeros, es decir, una colección con los elementos 2, 20, 30, 12 y 84.

Volviendo a `#cantidadMaximaDePasajeros`, para terminar le enviamos el mensaje **`#sum`** a la colección de cantidades que me devolvió `#collect:`. El mensaje #sum devolverá el resultado de sumar todos los elementos de una colección. Claramente, esto sólo funciona para colecciones con elementos numéricos.

	Cómo es el bloque?	Qué hace?	Qué retorna?	Cuántos elementos tiene la colección resultante?
colección **anySatisfy:** bloque	Un bloque que recibe **un parámetro**, que representa a cada elemento de la colección, y **devuelve un booleano**	Retorna true **si al menos un elemento de la colección cumple el bloque**; sino devuelve false	**true / false**	no devuelve una colección!
colección **allSatisfy:** bloque	Un bloque que recibe **un parámetro**, que representa a cada elemento de la colección, y **devuelve un booleano**	Retorna true **si todos los elementos de la colección cumplen el bloque**; sino devuelve false	**true / false**	no devuelve una colección!
colección **select:** bloque	Un bloque que recibe **un parámetro**, que representa a cada elemento de la colección, y **devuelve un booleano**	Retorna una **nueva colección** que tiene agregados **los elementos de la colección original que cumplen el bloque**	una **nueva colección que tiene elementos del mismo tipo** que la colección original	la **nueva colección** tiene una **cantidad de elementos ≤** que la cantidad de elementos **que tiene la colección original**
colección **collect:** bloque	Un bloque que recibe **un parámetro**, que representa a cada elemento de la colección, y **devuelve un objeto**	Retorna una **nueva colección** que tiene agregados todos **los objetos que fueron resultado de evaluar el bloque con cada elemento de la colección original**	una **nueva colección que no** necesariamente **tiene elementos del mismo tipo** que la colección original	la **nueva colección** tiene una **cantidad de elementos =** que la cantidad de elementos **que tiene la colección original**