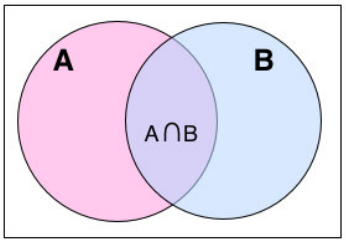
TIPOS DE COLECCIONES

Collection: es una clase abstracta que provee métodos default para usar sobre colecciones (size, select:, collect:, inject:into:)

Bag: una bolsa/valija donde guardamos cosas.
Como toda valija (especialmente la mía) las cosas están desordenadas, si busco algo tengo que empezar a sacar todo hasta que lo encuentro.
Sirve para representar un conjunto con elementos repetidos, una implementación posible sería usarlo como un carrito de compras en el supermercado.

Set: es un conjunto matemático, pero sin repetidos.
No hay orden (no tiene sentido en esta abstracción).
Un uso posible es si tengo las ventas del mes y quiero traer los clientes que compraron (pero sin traerlos repetidos por cada venta):

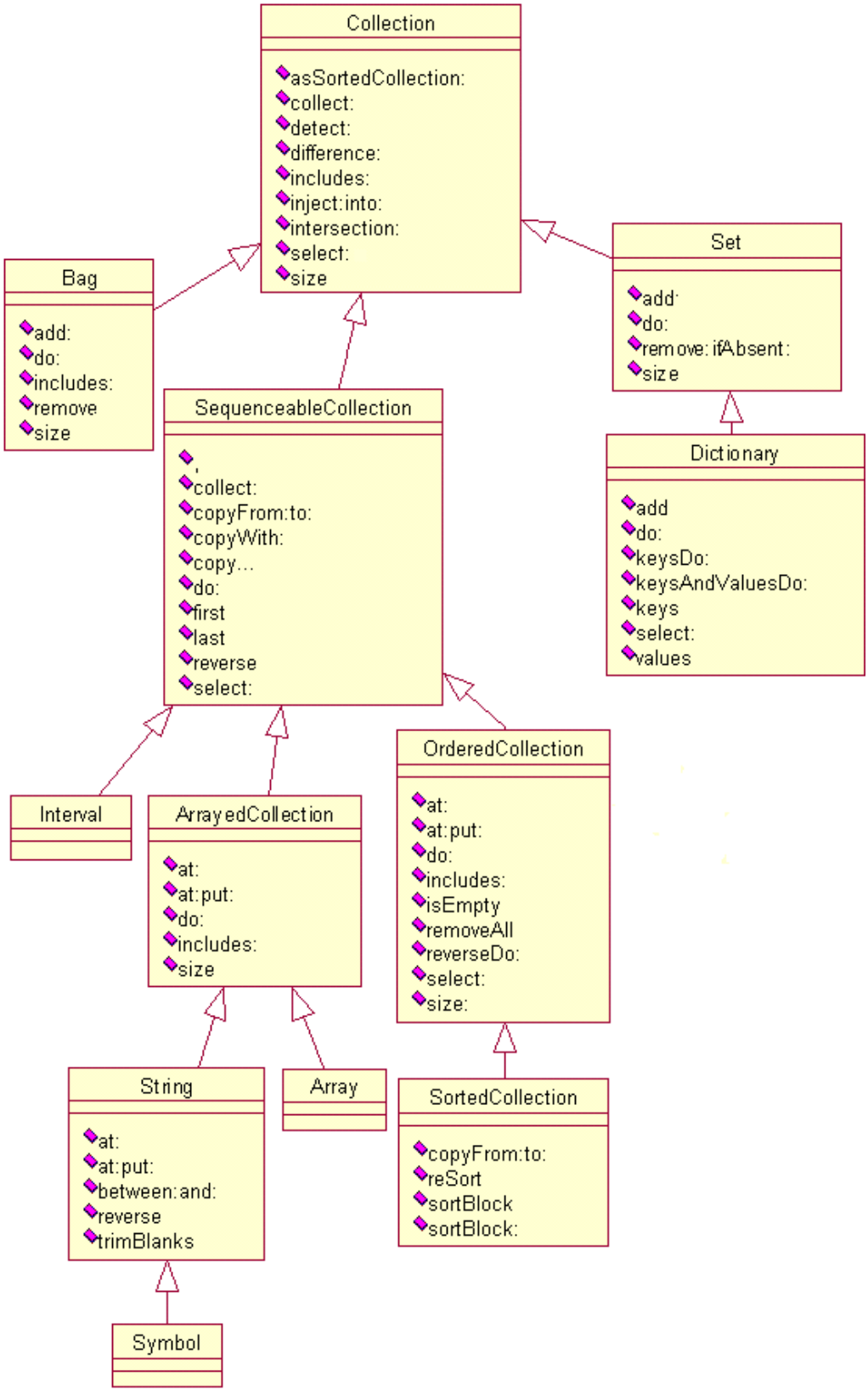


…
(ventas collect: [:venta | venta cliente]) asSet

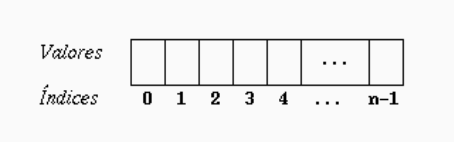
Dictionary: permite armar un par clave-valor.
La típica implementación es una agenda de teléfonos.

- Clave: 'Cynthia López'
- Valor: '4101-2992'

Restricción: la clave no puede repetirse.
La forma estándar de acceder a los elementos es por la clave (agenda at: 'Carla Conte') …
Pero también se proveen mensajes para acceder secuencialmente a las claves y a los valores

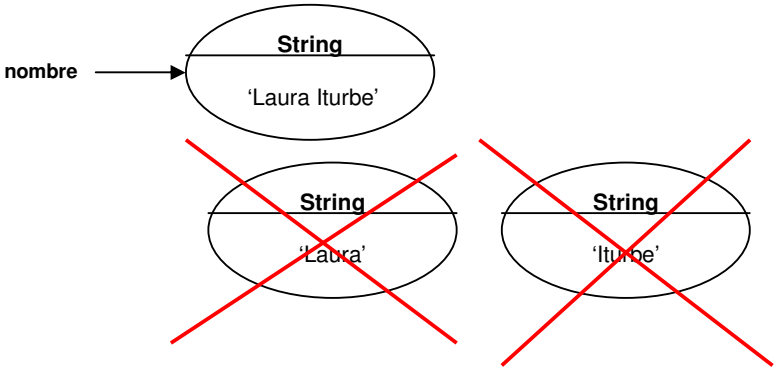

Collection
asSortedCollection:
collect:
detect:
difference:
includes:
inject:into:
intersection:
select:
size
Bag
add:
do:
includes:
remove
size
Set
add:
do:
remove:ifAbsent:
size
SequenceableCollection
,
collect:
copyFrom:to:
copyWith:
copy...
do:
first
last
reverse
select:
Dictionary
add
do:
keysDo:
keysAndValuesDo:
keys
select:
values
Interval
ArrayedCollection
at:
at:put:
do:
includes:
size
OrderedCollection
at:
at:put:
do:
includes:
isEmpty
removeAll
reverseDo:
select:
size:
String
at:
at:put:
between:and:
reverse
trimBlanks
Array
SortedCollection
copyFrom:to:
reSort
sortBlock
sortBlock:
Symbol

COLECCIONES ESTÁTICAS

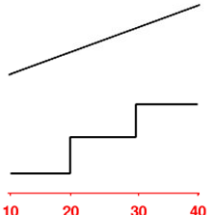


Array: el viejo y querido vector. La cantidad de elementos permanece fija desde la creación de un Array:
diasDeLaSemana ← Array new: 7.
Puedo acceder a los elementos a través de un índice
diasDeLaSemana at: 1
(pero no por clave). El orden está dado por la posición que ocupa cada elemento en el array.
El array es estático, me sirve si conozco de antemano la cantidad de elementos.

String: se implementan como una colección fija de caracteres.
Un String es inmutable, una vez que se crea no cambia su estado.
Si evaluamos, línea por línea:
nombre ← 'Laura'.
nombre ← nombre , ' Iturbe'.
Lo que ocurre es que el String 'Laura' se concatena con otro String: 'Iturbe' para formar un nuevo String: 'Laura Iturbe'. ¿Qué pasa con los String 'Laura' e 'Iturbe' una vez que pasa el Garbage Collector? Se los lleva…



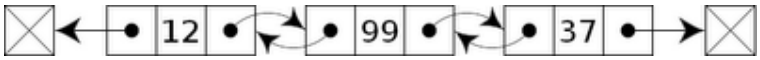
Interval: representan un intervalo cerrado de números enteros.
[1, 5] en matemática, se crea en Smalltalk
mediante
1 to: 5 → devuelve un Interval, al que le puedo pedir
size, do:, etc.



COLECCIONES DINÁMICAS

Además del Set, Bag y Dictionary tenemos:

OrderedCollection: es una lista que respeta el mismo orden en que fui insertando cada elemento (aunque puedo insertar o borrar elementos que están antes o después de otro).



SortedCollection: es una colección ordenada por un criterio.
¿Puedo ordenar una colección que tiene tanto clientes como proveedores?
Sí, si ambos tienen al menos un mensaje en común.
Por ejemplo los ordeno por razón social…
Como agregar nuevos elementos en una SortedCollection suele tener su costo (ya que tiene que mantener la colección ordenada), en general se utiliza una OrderedCollection (o un Set, o un Bag, etc.) para almacenar los clientes y cuando necesitamos listar los clientes por deuda (primero el que me debe más), genero una SortedCollection temporal, de la siguiente manera:

clientesPorDeuda (#AnalizadorMorosos)

```
^self clientes asSortedCollection: [ :a :b | a deuda > b deuda ]
```

a y b son… clientes (u objetos que entienden el mensaje *deuda*)

Si tienen una clase Cliente que entiende los mensajes deuda: y deuda, pueden probar en un workspace:

```
renault := Cliente new deuda: 90.
volkswagen := Cliente new deuda: 150.
[ :a :b | a deuda > b deuda ] value: renault value: volkswagen
```

¿Qué devuelve?

Otras colecciones para chusmear en casa:

- **ReadStream** / **WriteStream**

Y por último, presentamos al do:, que aplica una operación a cada uno de los elementos de una colección:

Si quiero patear todas mis entrevistas de trabajo dos días más tarde:

```
entrevistas do: [ :each | each postergar: 2 ]
```

Si quiero sacar 100 pesos de todas mis cuentas bancarias:

```
cuentasBancarias do: [ :each | each retirar: 100 ]
```

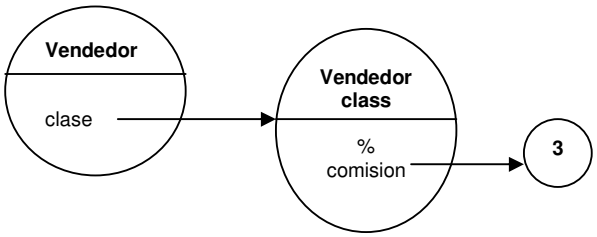
La operación que le pasamos es un bloque que recibe un argumento (que se aplica elemento a elemento de la colección).

VARIABLES Y MÉTODOS DE CLASE

Hasta ahora nosotros le enviamos mensajes a un objeto, que es instancia de una clase. Pero muchas veces nos pasa que en el requerimiento tenemos información que es compartida por todos los objetos de una misma clase.

- El porcentaje de comisión para todos los vendedores es de un 3%
- Un cliente puede subirse a cualquier juego de montaña rusa hasta 5 veces
- Todos los alumnos promocionan las materias con una nota mínima de 8.

Entonces, no voy a poner el % de comisión en Vendedor, sino en la clase Vendedor. Gráficamente:



En Smalltalk hay un objeto que representa la clase, que es donde están almacenados los métodos. También puedo guardar variables, que voy a llamar **variables de clase**.
Cuando yo defino una clase, puedo definir ahí cuáles son sus variables de clase:

```
Object subclass: #Vendedor
    instanceVariableNames: 'nombre legajo'
    classVariableNames: 'PorcentajeComision'
    poolDictionaries: ''
    classInstanceVariableNames: ''
```

Las variables de clase comienzan con mayúscula, porque son globales para todas las instancias de esa clase (cualquier objeto de esa clase tiene acceso a esa variable).

registrarComisionPara: unaVenta (#Vendedor)
  self totalComisiones: unaVenta montoTotal * PorcentajeComision

¿Y los métodos de clase?
Son los métodos en los que el objeto receptor es la clase.
Ya conocen uno: new.

Vendedor new ← devuelve una nueva instancia de Vendedor

¿Qué otro método podríamos definir?
porcentajeComision (MC de #Vendedor)
  ^PorcentajeComision

¿Entonces qué podríamos cambiar en registrarComisionPara:? Podríamos usar acceso indirecto:

registrarComisionPara: unaVenta (#Vendedor)
  self totalComisiones: unaVenta montoTotal * self class porcentajeComision

Desde un método de instancia *self class xxxxxx* envía un mensaje a la clase Vendedor (Vendedor class).

Siempre que haya varias instancias compartiendo información → puedo abstraer una variable de clase y accederla mediante métodos de clase.

¿Para qué otra cosa sirve tener un método de clase? Para mejorar la semántica de los métodos. Si yo estoy en un workspace, puedo hacer:

Cliente new nombre: 'Laura'.

O también:

Cliente deNombre: 'Laura'.

¿Implementamos el método de clase deNombre:?

deNombre: unNombre (MC #Cliente)
  ^self new nombre: unNombre

self apunta al objeto receptor. Si el método es de clase, self apunta a una clase.

¿Qué gano generando un nuevo método y delegando al new y a los setters? Gano en que desde el workspace lo que escribo se parece más a castellano y queda más claro. Pero también gano otra cosa: si se fijan en la clase Point, van a ver que el mensaje x: y: reemplaza al new.

Point x: 2 y: 3. ¿Por qué? Hay dos razones válidas para modificar el new:

- Por la claridad para quien lo usa y todo lo que dijimos recién
- Porque la gente que hizo la clase Point consideró que para poder usar un objeto punto tiene que estar inicializado correctamente, con la abscisa y la ordenada informada. Entonces es común que cuando se cree el objeto también se inicialice para poder usarlo sin problemas.