Universidad Tecnológica Nacional

Facultad Regional Buenos Aires

Cátedra de Paradigmas de Programación

Colecciones en Smalltalk

Autores:

- Victoria Pocladova – pocladova@yahoo.com.ar
- Carlos Lombardi – carlombardi@gmail.com
- Leonardo Volinier – leonardo.volinier@gmail.com
- Jorge Silva – josilva@gmail.com

Historial de Revisiones

Versión	Revisado por	Fecha	Detalles
V 0.1	Vicky		Versión Inicial, con algunas observaciones y temas a incluir
V 0.2	Carlos		Algunas correcciones, se agregaron secciones “Colecciones y referencias” y “Navegando en colecciones”
V 0.3	Vicky	05/09/2006	Retoqué la parte de Introducción, y “Que podemos hacer con una colección?”
V 0.4	Carlos	13/09/2006	Retoques finales, organización del contenido de “Colecciones y referencias”.
V 0.5	Carlos	14/09/2006	Se libera la primer versión, hasta “Colecciones y referencias” inclusive.
V 0.6	Leonardo	03/04/2007	Se agrega la sección de “Sabores de colecciones”
V 0.7	Carlos	09/04/2007	Retoques y subsecciones en “Sabores de colecciones”
V 1.0	Jorge – Carlos	21/04/2008	Secciones “Hablando con los elementos” y “Resumen de mensajes que entiende una colección”

Colecciones en Smalltalk

Introducción

Las colecciones nos resultan de utilidad porque nos permiten agrupar objetos, para luego poder operar sobre un elemento en particular, sobre algunos elementos seleccionados mediante un filtro, o sobre una colección como conjunto.

Esto nos permite modelar conjuntos o agregados de cosas, que son muy comunes en casi todos los dominios en los que podemos pensar: las piezas de un tablero de ajedrez, los integrantes de un equipo de fútbol, las líneas de una factura, ¡de todo!

A primera vista una colección es un conjunto de objetos.
Si la vemos con más precisión nos damos cuenta que es más preciso pensarla como un conjunto de referencias: los elementos no están adentro de la colección, sino que la colección los conoce.

A su vez, como todo en Smalltalk es un objeto, podemos deducir que una colección también es un objeto.

También sabemos que en Smalltalk todo objeto es instancia de una clase, entonces va a haber una clase Collection. En realidad hay toda una jerarquía con distintos "sabores", distintos tipos de colecciones que nos van a servir para distintos fines.

Este apunte se complementa con la guía de lenguajes, en el que se detallan varios mensajes que entienden las colecciones.

¿Qué podemos hacer con una colección?

Para tratar de responder esta pregunta no es necesario estar familiarizado con las colecciones de Smalltalk. Recordemos que una de las características ventajosas del Paradigma de Objetos es que acerca nuestro modelado a la forma en que percibimos la realidad.

Entonces ¿que haríamos nosotros con una colección?

Pensemos en una colección de estampillas.

Podemos:

Mirarlas → recorrer la colección
Encuadernarlas por fecha → ordenar la colección
Conseguir nuevas estampillas → agregar elementos a la colección
Regalar una estampilla → quitar elementos de la colección.
Quedarme con las estampillas de Brasil → hacer un filtro o selección de los elementos según un criterio.
Saber si tengo una determinada estampilla → preguntarle a una colección si incluye o no un determinado objeto como elemento.

¿Qué otra colección podemos modelar del mundo real?
¡¡Un carrito del supermercado!!

Mientras vamos de góndola a góndola del super vamos agregándole elementos (referencias a objetos) a esa colección.

Cuando llegamos a la caja recorremos esa colección y obtenemos información de la misma: cuanto suma el costo total de sus artículos, cuántos artículos compré …

¿Ahora, como podemos modelar todas estas situaciones con objetos usando Smalltalk?... ¡Enviando mensajes a colecciones!

Recordemos que en el paradigma de objetos todo programa es un conjunto de objetos enviándose mensajes para concretar un objetivo en común. Bien, en este caso los objetos serán las distintas colecciones y sus elementos. Los mensajes que puede recibir una colección serán, entre otros, los que mencionamos para la colección de estampillas, dándoles un toque smalltalkero ;-)

Un ejemplo rápido

La primer clase que vamos a usar y que implementa la idea de colección se llama Set. Entonces, para crear una colección y asignarla a una variable, escribiremos

```
coleccionEstampillas := Set new.
```

Ahora definamos tres nuevas estampillas y agreguémoslas a la colección, enviándole el mensaje add:. Defino una cuarta estampilla que no agrego.

```
pilla := Estampilla new.
pilla origen: 'Brasil'.
pilla alto: 6 ancho: 3.
pilla2 := Estampilla new.
pilla2 origen: 'Alemania'.
pilla2 alto: 4 ancho: 2.
pilla3 := Estampilla new.
pilla3 origen: 'Brasil'.
pilla3 alto: 5 ancho: 3.
pilla4 := Estampilla new.
pilla4 origen: 'Brasil'.
pilla4 alto: 3 ancho: 1.
coleccionEstampillas add: pilla.
coleccionEstampillas add: pilla2.
coleccionEstampillas add: pilla3.
```

A esta colección ya le puedo hacer algunas preguntas

```
coleccionEstampillas size.    "devuelve 3"
coleccionEstampillas includes: pilla2.   "devuelve true"
coleccionEstampillas includes: pilla4.   "devuelve false"
```

Claro que para que la colección me sea realmente útil, me debe permitir interactuar con sus elementos, poder hablarle (p.ej.) a pilla2 a través de la colección.

Antes de ver cómo hacer esto, clarifiquemos un poco la relación entre una colección y sus elementos.

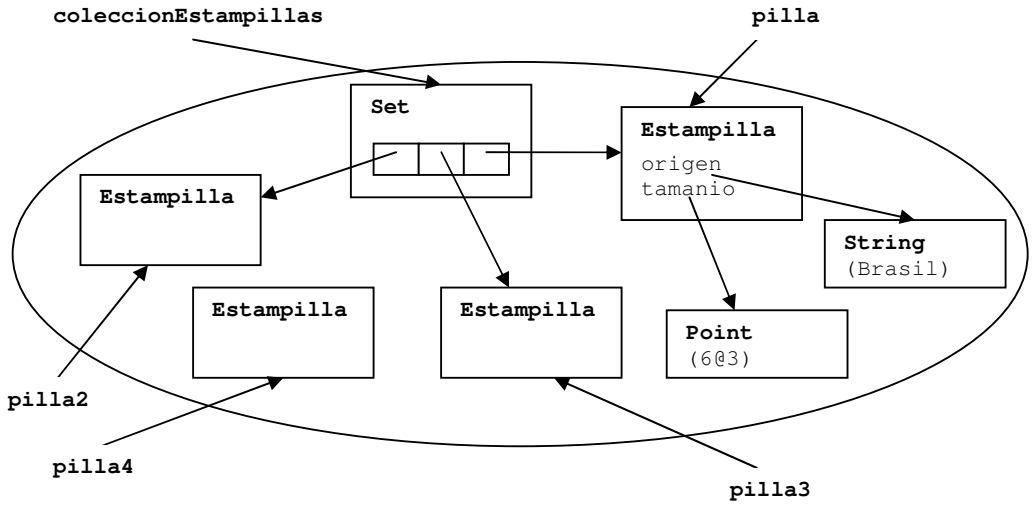
Colecciones y referencias

Como ya dijimos en la introducción, si miramos una colección en detalle, vemos que es mejor entenderla como un conjunto de referencias a objetos, y no como un conjunto de objetos.

¿Cuál es la diferencia? Que conceptualmente, la colección no tiene "adentro suyo" a sus elementos.

Los elementos de una colección son objetos como cualesquiera otros, al agregarlos a una colección estoy, en realidad, agregando una referencia que parte de la colección y llega al objeto "agregado". Los objetos que elijo agregar una colección pueden estar referenciados por cualesquiera otros objetos.

En el ejemplo anterior, algunas de las estampillas que creamos son elementos de la colección, y además están referenciadas por variables. Gráficamente tenemos



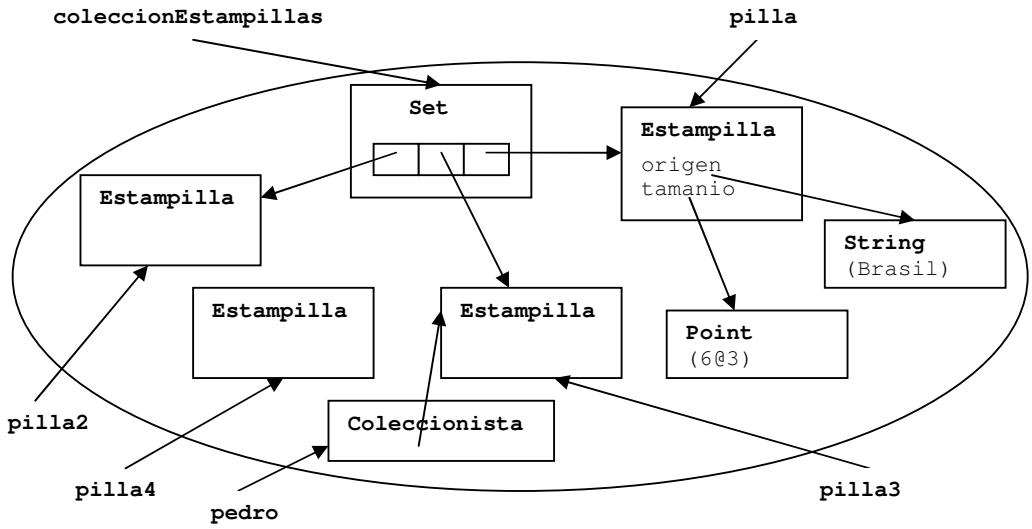
Tres aclaraciones sobre el gráfico:

1. por una cuestión de espacio detallamos el estado interno sólo para una estampilla, está claro que todas las estampillas conocen a un String que representa su origen y a un Point que representa su tamaño.
2. la clase Point sí existe en Dolphin (y en todos, o la gran mayoría, de los ambiente Smalltalk) y sirve para representar p.ej. puntos en coordenadas de dos dimensiones o superficies rectangulares.
3. ¿en qué orden "están" las estampillas en el Set? Un Set no mantiene sus elementos en un orden determinado, más adelante veremos que hay distintos "sabores" de colecciones, algunos mantienen orden y otros no.

Ahora agreguemos un objeto más

```
pedro := Coleccionista new.
pedro estampillaPreferida: pilla3.
```

El ambiente queda así:



(omitimos el nombre de la variable del coleccionista también por cuestiones de espacio)

Ahora hay un objeto que tiene 3 referencias:

1. es el objeto referenciado por la variable pilla3
2. es un elemento del Set
3. es la estampilla preferida del Coleccionista

Ya podemos ver un poco más en detalle la relación entre una colección y sus elementos. La colección maneja referencias a los elementos que le voy agregando (p.ej. enviándole el mensaje `add:`), análogas a las referencias de las variables de instancia de otros objetos[1]. Así, los objetos que quedan referenciados por la colección pueden tener otras referencias sin problema.

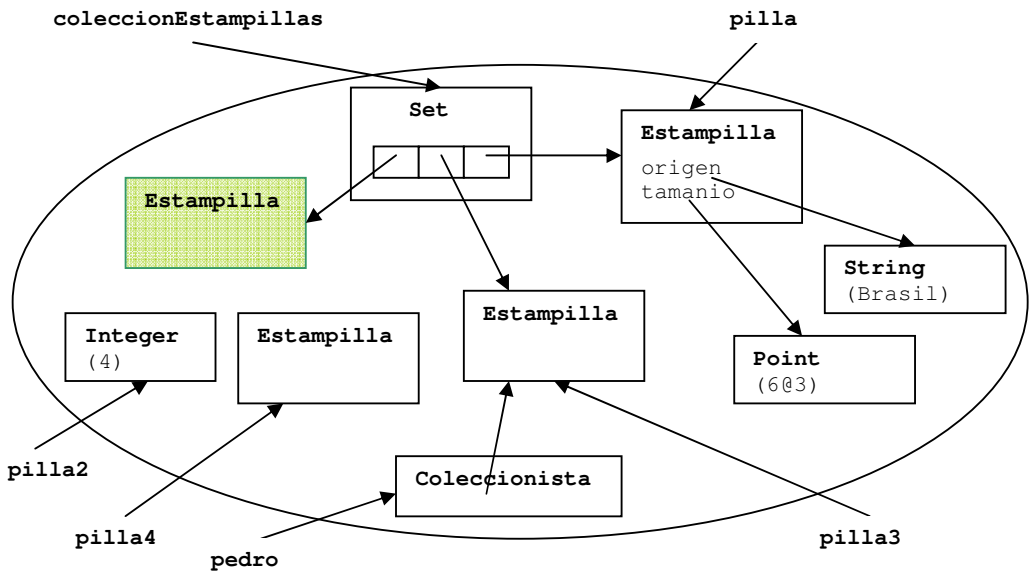
Un objeto no tiene nada especial por ser elemento de una colección, sólo una referencia más hacia él. Un objeto no conoce, en principio, de qué colecciones es elemento (podría tener una referencia explícita a la colección, pero eso habría que programarlo a mano).

La referencia a un objeto por ser elemento de una colección cuenta para que el objeto no salga del ambiente cuando pasa el Garbage Collector. Veámoslo con un ejemplo, agregando esta línea de código:

[1] Hay dos diferencias entre las referencias que mantiene un Set y las que mantiene p.ej. una Estampilla
Cada referencia de la Estampilla tiene un nombre, que es el nombre de la variable de instancia; las del Set son anónimas.
Cada estampilla tiene una cantidad fija de referencias (son siempre 2), un Set puede tener una cantidad arbitraria, que crece a medida que le agrego elementos al Set.

```
pilla2 := 4
```

El ambiente queda así:



La estampilla que marcamos en verde ya no está referenciada por la variable pilla2, pero sigue viva en el ambiente, porque tiene la referencia del Set.

Hablando con los elementos

Hay algunas operaciones que se hacen sobre una colección, en la que parte de lo que hay que hacer, es responsabilidad de cada elemento.

Por ejemplo, supongamos que quiero obtener, de mi colección de estampillas, aquellas que tengan más de 10 cm^2 de superficie.

La colección no sabe la superficie de cada estampilla, sí conoce a las estampillas, entonces puede enviarle mensajes a cada una. Lo que no sabe es qué mensajes puede enviarle, un Set no sabe si lo que tiene son estampillas, perros, números, otros Set, o cualquier otro objeto, sólo representa al conjunto, sin saber nada de sus elementos.

Por otro lado, cada estampilla no sabe en qué colección está, de hecho un mismo objeto podría estar en varias colecciones.

Por lo tanto, para resolver mi problema necesito que actúen tanto la colección (que es la que conoce a los elementos) como cada elemento (que es el que sabe su superficie). Veamos cómo lograr esta interacción.

Empecemos por decidir a quién le pedimos lo que queremos. Quiero aquellas estampillas, de las que son elementos de `coleccionEstampillas`, que cumplan una determinada condición.

¿A qué objeto le voy a pedir esto? A la colección.

El selector (nombre del mensaje) es `select:` …

… o sea que necesita un parámetro. Este parámetro va a representar la condición, que es un código que se va a evaluar sobre cada elemento, y debe devolver true o false.

Los objetos que representan "cachos de código" son los bloques[2], en este caso un bloque con un parámetro. Queda así

```
coleccionEstampillas select: [:estam | estam superficie > 10]
```

Veamos "cómo es que funciona"

El bloque es el que sabe qué preguntarle a cada estampilla, representa la condición. Cambiemos el código anterior un poco

```
condicion := [:estam | estam superficie > 10].
coleccionEstampillas select: condicion.
```

Ahora el bloque (que es un objeto, tan objeto como el que representa una estampilla, o el que representa un conjunto) está referenciado por una variable. Entonces puedo evaluar, por ejemplo, la condición para pilla. Los bloques entienden el mensaje value:. Lo escribo así:

```
condicion value: pilla
```

si evalúo esto, devuelve true.

La `coleccionEstampillas` es la que conoce a sus elementos.
Sabe que cuando le llega el mensaje **select:** con el **bloque** de parámetro, lo que tiene que hacer es evaluar ese **bloque** con cada uno de sus elementos (los elementos de `coleccionEstampillas`). Esta es la parte que maneja la colección.

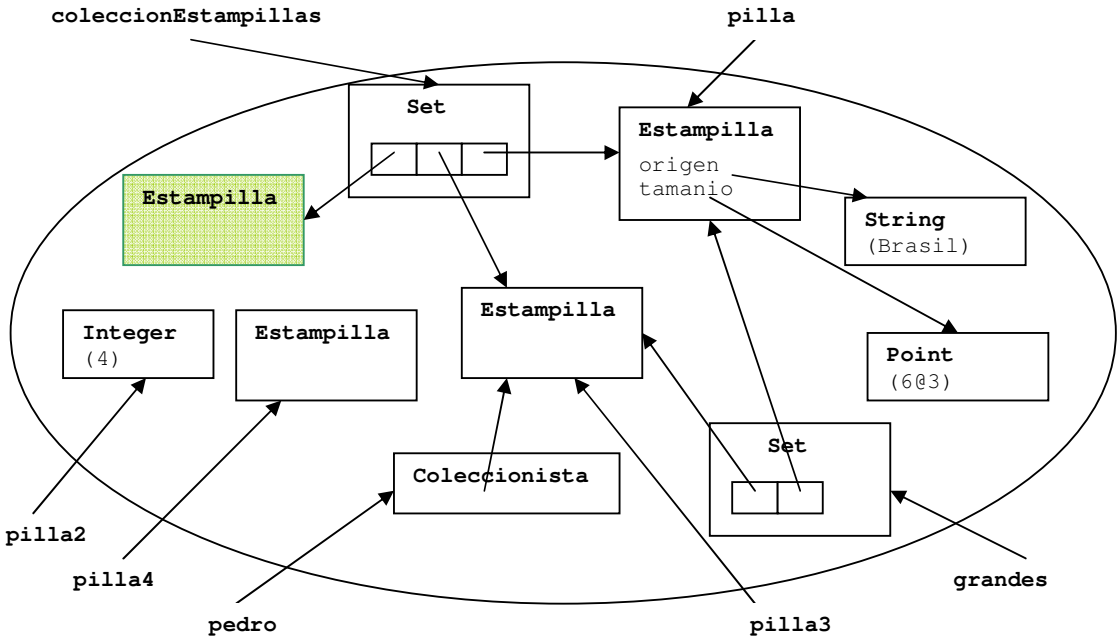
Las estampillas entienden el mensaje `superficie`, eso es lo que sabe hacer cada estampilla, devolver su superficie.

Finalmente, ¿qué devuelve esto?
Otra colección, distinta de `coleccionEstampillas`, y que tiene como elementos algunos de los objetos que también tiene `coleccionEstampillas` como elementos; exactamente los que cumplen la condición que le paso como parámetro. Si pongo

```
grandes := coleccionEstampillas select: [:estam | estam
superficie > 10]
```

el ambiente va a quedar así

[2] Ver apunte sobre bloques … que agregaremos en algún momento.



En este diagrama, podemos ver

1. que el resultado del select: es una colección distinta de `coleccionEstampillas`
2. que `coleccionEstampillas` no se modificó como resultado del select:
3. que hay dos estampillas que son elementos de dos colecciones.

Ahora bien, si agregamos a `coleccionEstampillas` una nueva estampilla grande, ¿se agrega también en la colección referenciada por `grandes`?
No, porque `grandes` se creó en el `select:` y está separada de `coleccionEstampillas`.

En resumen: cuando quiero hacer una operación sobre una colección que necesita enviarle mensajes a cada elemento, la operación se la pido a la colección, y le voy a enviar como parámetro un bloque que describe la interacción con cada elemento.

Hay varias operaciones de este estilo,

Sabores de colecciones

En la introducción de esta obra, mencionamos que Smalltalk tiene una clase llamada Collection que me permite representar colecciones, pero Collection es la abstracción superior de la jerarquía de colecciones. Al momento de instanciar una colección, tengo que decidirme por un sabor. En el “ejemplo rápido” usamos Set, que es uno de esos sabores, pero hay más.

A continuación intentaremos mostrar los sabores de colecciones, tratando de ordenarlos en base a sus respectivos grados de especialización.

Set y Bag

El primer sabor que visitaremos es el Bag. Esta colección es de las denominadas "sin orden" y como su nombre lo indica, es la representación de una bolsa. Una bolsa de objetos. Por ejemplo, al principio, hablamos de un carrito de compras. El Bag la colección más adecuada para esta representación. Imagínense que puedo poner tres latas de tomates, dos botellas de agua y siete peras.

El segundo sabor que veremos es el Set (si, el que está más arriba en el ejemplo). El Set, como su nombre lo indica, está concebido para la representación de conjuntos. Una propiedad, fundamental diría, sobre conjuntos es que los elementos que pertenecen al conjunto son únicos en él. En resumidas palabras, en los conjuntos (Sets) no voy a tener dos veces el mismo elemento. O sea el Set no admite repetidos. Salvo por esta propiedad, el comportamiento es el mismo que el del Bag.

Volviendo al ejemplo del supermercado, imagínense que tengo una lata de tomates. Lo voy a representar así:

```
unaLata := LataDeTomates new.
unaLata conTomates: 'perita'.
unaLata deLaMarca: 'La Marca que a uds les guste'
unaLata conPrecio: 1.00
```

Bien, tengo mi objeto `unaLata` y quiero agregar a mi carrito de compras 3 latas de tomates. Entonces hago:

```
miCarrito add: unaLata.
miCarrito add: unaLata.
miCarrito add: unaLata.
```

En mi carrito debería haber 3 latas de tomates. Pero, como vimos, si represento a mi carrito de compras con un Bag, (o sea que hice `miCarrito:= Bag new`.) entonces sí voy a tener 3 objetos dentro de mi carrito.

Si representara mi carrito de compras con un Set (o sea que hice `miCarrito:=Set new`.) entonces hubiese tenido un solo objeto dentro de mi colección. Simplemente, cuando hago el segundo `miCarrito add: unaLata` el Set identifica que ya tiene ese objeto en la colección y no lo vuelve a agregar.

Colecciones ordenadas

Entonces ya vimos el Set y Bag, que son colecciones sin orden. Ahora introduciremos algunas colecciones con orden.

¿Qué quiere decir "ordenadas"? Que hay un elemento 1, un elemento 2, etc., al contrario de un Set o un Bag, en donde los elementos están "todos tirados".

O sea, puedo acceder a los elementos de cualquier colección ordenada (veremos que hay varias variantes) parecido a como se accede a un Array en C o Pascal.

Para eso le envío a la colección el mensaje `at:` , si quiero el cuarto elemento de miCol que es una colección ordenada, lo puedo pedir así:

```
miCol at: 4
```

y tengo los mensajes `first` y `last` que devuelven el primero y el último.

Eeeehhh pero … si tengo un Set o un Bag, ¿ cómo accedo a los elementos de un Bag o un Set? Eso … viene más adelante.

Veamos ahora las variantes de colecciones ordenadas.

La primera que les presentaremos es la OrderedCollection. Esta colección ordena sus elementos, y el criterio es el orden en cual fueron agregados a la colección.

Entonces si creo mi colección así

```
miCol := OrderedCollection new.
miCol add: 'hola'.
miCol add: 'queridos'.
miCol add: 'amigos'.
miCol add: 'escandinavos'.
```

después puedo pedirle varias cosas

`miCol first`	el primero	`'hola'`
`miCol last`	el último	`'escandinavos'`
`miCol at: 3`	el tercero	`'amigos'`
`miCol at: 4`	el cuarto	`'escandinavos'`
`miCol size`	cantidad de elementos	`4`

y después puedo seguir agregándole elementos, ante lo cual la colección "se estira"

```
miCol add: 'agrego'.
miCol add: 'cosas'.
```

`miCol size`	cantidad de elementos ahora	`6`
`miCol last`	el último ahora	`'cosas'`

Luego, así como la OrderedCollection, también tenemos la llamada SortedCollection, que la diferencia con la primera radica en que el criterio de ordenamiento puede ser definido. Si no definimos el criterio, SortedCollection ordena los elementos por su “orden natural” (significa que los ordenará de menor a mayor).

Dicho de otra forma, no ordena por orden de llegada, sino por comparación entre los elementos.

Si queremos ordenar los elementos de la colección con un criterio en particular, necesitamos pasárselo a la colección. La forma de hacerlo, es pasarle lo que denominamos “sortBlock”, que es un objeto Block (bloque). Este objeto lo veremos más adelante.

Con respecto a las colecciones que tienen orden, por último veremos al viejo amigo Array. Aquí en Smalltalk también existe, y una de sus características es que es de tamaño fijo (para instanciar un Array, hago `Array new: 6`, donde 6 es la cantidad de elementos que contendrá).

Los Arrays no implementan el mensaje `add:`, justamente porque no puedo modificar su tamaño. La forma de agregarles elementos es a través del mensaje `at:put:`, como por ejemplo:

```
miVector := Array new: 2.
miVector at: 1 put: unaLata.
```

Todas las colecciones entienden una serie de mensajes que permiten obtener distintos sabores de colecciones con los mismos elementos. Estos mensajes son de la forma “as{ColeccionQueQuiero}”. Vamos a un par de ejemplos para ver cómo funciona.

Si tuviese una colección de la clase Bag, y quiero sacarle los repetidos, sé que el Set no tiene repetidos, entonces tomo mi colección como un Set. Entonces:

```
sinRepetidos := miCarrito asSet.
```

Si tuviese un array, y lo quiero convertir en una colección de tamaño variable, podría hacer:

```
coleccionVariable := miVector asOrderedCollection.
```

Si quisiera ordenar mi carrito de compras del producto más caro al más barato, haría algo como:

```
ordenadosPorPrecio := miCarrito asSortedCollection:
          [:unProd :otroProd | unProd precio > otroProd precio].
```

El mensaje `asSortedCollection:` recibe un parámetro que, obviamente, es un sortBlock.

El sortBlock es un bloque que necesita 2 parámetros. El código del bloque es un código que debe devolver true o false. Para ordenar los objetos dentro de la colección, se evalúa el código y si el objeto retornado es true, el primer parámetro va antes que el segundo. Si retorna false, el segundo parámetro se ubica antes que el primero.

También está el mensaje `asSortedCollection` (o sea sin el dos-puntos, o sea que no requiere parámetro) que, como dijimos antes, ordenará los elementos por el “orden natural”.

¿Cuál es el "orden natural"?

Dijimos que si a una SortedCollection no le decimos cómo queremos que ordene los elementos, los ordena por el "orden natural". Pero … ¿qué puede ser este "orden natural"?

Si estoy en el paradigma de objetos … seguro va a tener que ver con objetos y mensajes. El "orden natural" es el que dicta el mensaje `<`, que es binario.

O sea, en una SortedCollection con "orden natural" el criterio es poner a elem1 antes que elem2 si el resultado de evaluar

```
elem1 < elem2
```

es `true`.

Claro, eso quiere decir que solamente voy a poder tener, en una SortedCollection con "orden natural", objetos que entiendan el mensaje <. Los números, los String, las fechas, todos esos entienden <. Pero p.ej. si quiero poner latas en una SortedCollection, no puede ser por "orden natural", tengo que especificar el orden con el bloque con dos parámetros como vimos hace un ratito.

Diccionarios

Por último, queremos mostrarles un sabor de colección que es especial. Se llama Dictionary y, como su nombre lo indica, intenta representar un diccionario. Este tipo de representación implica tener una asociación entre una clave y un valor.

Poniendo como ejemplo, el propio diccionario. El diccionario es una asociación entre una clave, que son cada letra del abecedario y un apartado de páginas que tienen palabras.

El diccionario lleva todo una explicación aparte, por lo que decidimos ponerlo en un apunte aparte. Busquen el apunte sobre Diccionarios.

Resumen de mensajes que entiende una colección

Lo que sigue sirve para todos los sabores de colecciones, más información en la guía de lenguajes.

¿Qué puedo hacer con los conjuntos de objetos?

Agregarle objetos

¿Que mensaje envío? add:
Un ejemplo: `pajaros add: pepita.`
Otro ejemplo: `usuarios add: usuario23.`

Quitarle objetos

¿Que mensaje envío? remove:
Un ejemplo: `pajaros remove: pepita.`
Otro ejemplo: `usuarios remove: usuario23.`

Preguntarle si tiene un objeto

¿Que mensaje envío? includes:
¿Que me devuelve? Un objeto booleano, true o false
Un ejemplo: `pajaros includes: pepita.`
Otro ejemplo: `usuarios includes: usuario23.`

Unirlo con otro conjunto

¿Que mensaje envío? union:
¿Que me devuelve? Una nueva colección con la unión de ambas.
Un ejemplo: `golondrinas union: picaflores.`
Otro ejemplo: `mujeres union: hombres.`

Seleccionar los elementos que cumplen con un criterio

¿Que mensaje envío? select:
¿Que me devuelve? Una nueva colección con los objetos que cuando se los evalúa con el bloque, dan true.
Un ejemplo: `pajaros select: [:unPajaro | unPajaro estaDebil ].`
Otro ejemplo: `usuarios select: [:unUsuario | unUsuario deuda > 1000].`

Seleccionar los elementos que no cumplen con un criterio

¿Que mensaje envío? reject:
¿Que me devuelve? Una nueva colección con los objetos que cuando se los evalúa con el bloque, dan false.
Un ejemplo:

```
pajaros reject:
      [:unPajaro | (unPajaro estaDebil | unPajaro estaExcitado)].
```

Otro ejemplo:

```
 usuarios reject: [:unUsuario | unUsuario esDeudor].
```

Recolectar el resultado de hacer algo con cada elemento

¿Que mensaje envío? collect:
¿Que me devuelve? Una nueva colección con los objetos que devuelve el bloque.
Un ejemplo:

```
      pajaros collect: [:unPajaro | unPajaro ultimoLugarDondeFue].
```

Otro ejemplo: `usuarios collect: [:unUsuario | unUsuario nombre].`

Verificar si todos los elementos de la colección cumplen con un criterio

¿Que mensaje envío? allSatisfy:
¿Que me devuelve? True, si todos los objetos de la colección dan true al evaluarlos con el bloque.
Un ejemplo: `pajaros allSatisfy: [:unPajaro | unPajaro estaDebil].`
Otro ejemplo:

```
      usuarios allSatisfy: [:unUsuario | unUsuario gastaMucho].
```

Verificar si algún elemento de la colección cumple con un criterio

¿Que mensaje envío? anySatisfy:
¿Que me devuelve? True, si alguno de los objetos de la colección da true al evaluarlo con el bloque.
Un ejemplo: `pajaros anySatisfy: [:unPajaro | unPajaro estaDebil].`
Otro ejemplo:

```
      usuarios anySatisfy: [:unUsuario | unUsuario gastaMucho].
```

Evaluar un bloque de los parámetros con cada elemento de la colección, usando como primer parámetro la evaluación previa, y como segundo parámetro ese elemento.

¿Que mensaje envío? inject:into:
¿Que me devuelve? La ultima evaluación del bloque.
¿Para qué me sirve? Para muchas cosas: obtener el que maximice o minimice algo, obtener el resultado de una operación sobre todos (p.ej. sumatoria), y más.

Un ejemplo (sumatoria):

```
pajaros
    inject: 0
    into: [:inicial :unPajaro | inicial + unPajaro peso].
```

Otro ejemplo (recolecto colección de colecciones):

```
#( 1 2 3 )
    inject: Set new
    into: [:divisores :nro | divisores union: (nro divisores)]
```

Y otro más (el que tiene más energía):

```
pajaros
    inject: (pajaros anyOne)
    into: [:masFuerte :unPajaro |
    (unPajaro energia < masFuerte energia)
         ifTrue: [unPajaro] ifFalse: [masFuerte]]
```