Colecciones en Smalltalk[1]

Introducción

Imaginemos querer modelar una agenda para una semana. Creamos nuestra clase Semana y, como sabemos que los días de la semana son 7, creamos 7 variables de instancia para cada uno de ellos en la clase. Esta situación no siempre es posible representar sólo con variables de instancias. Muchas veces vamos a necesitar modelar relaciones entre un objeto y varios (una relación 1 a N). Por ejemplo, los departamentos de un edificio, los estudiantes de una universidad, etc. En muchas oportunidades no vamos a poder conocer el número total o máximo de elementos o simplemente sólo sabremos que van a ser muchos. Para tales situaciones nos van a servir las colecciones.

Las colecciones permiten agrupar objetos para poder operar sobre ellos. En general podremos referirnos a esa colección de objetos bajo un mismo nombre, y poder agregar o sacar objetos, filtrarlos, recorrerlos y muchas otras cosas de manera mucho más sencilla. Las colecciones nos van a permitir modelar conjuntos de objetos, como ser: las piezas de un tablero, los empleados de una fábrica, la flota de buques de una empresa de transporte fluvial, etc. y poder realizar operaciones sobre ese grupo.

Las colecciones se pueden ver como un conjunto de objetos, y de hecho vamos a decir eso normalmente (abusando del lenguaje), pero en realidad las colecciones son conjuntos de referencias a objetos; es decir, los objetos no están adentro de la colección, sino que la colección los conoce (los referencia) de alguna forma.

Pero, ¿y si las colecciones son un conjunto de objetos (o referencias más correctamente), cómo las representamos? Sabemos que en Smalltalk sólo tenemos objetos y mensajes, por lo cual las colecciones van a ser representadas como... objetos!!!

Resumiendo, **un objeto colección es un objeto que representa un conjunto de referencias a otros objetos**, con el cual se van a poder realizar distintas operaciones sobre los elementos que ella referencia.

La pregunta que nos surge ahora es la que da título a la siguiente sección:

¿Qué operaciones podemos hacer con una colección?

Para poder responder más fácilmente y poder comprenderlo, vamos a elegir un ejemplo: una colección de CDs.

Con la colección de CDs podemos:

- Mirar los CDs → recorrer la colección
- Organizarlos por artista → ordenar la colección

---

1 *Basado en el apunte "Colecciones en Smalltalk" de la Universidad Tecnológica Nacional – Facultad Regional Buenos Aires (cátedra de Paradigmas de Programación), de Victoria Pocladova, Carlos Lombardi, Leonardo Volinier y Jorge Silva.*

- Agregar nuevos CDs → agregar elementos a la colección
- Regalar un CD → quitar elementos de la colección
- Quedarme con los CDs de rock nacional → hacer un filtro o selección de los elementos según un criterio.
- Saber si tengo un CD de Kill Bill → preguntarle a una colección si incluye o no un determinado objeto como elemento.

¿Y cómo se modela este comportamiento con objetos de Smalltalk? La respuesta debería ser obvia: con métodos definidos para las colecciones. Es decir, podemos crear nuestra instancia de colección que representa a nuestra colección de CDs y a ese objeto enviarle mensajes para agregarle un nuevo objeto CD, para que nos filtre los CDs de rock nacional, etc.

Primer ejemplo

La primera clase que vamos a usar es una colección que se llama `Set`. Un `Set` representa un conjunto entendido como los conjuntos matemáticos que aprendimos en la escuela.

Para crear una colección `Set` y asignarla a una variable que represente a nuestra colección de CDs, escribimos:

```
coleccionCDs := Set new.
```

Supongamos que tenemos definida la clase de los CDs. Creamos instancias de CDs y los agregamos a nuestro conjunto `coleccionCDs`, mandándole el mensaje `add:`. Definimos además un cuarto CD que no agregamos.

```
cd1 := CD new.
cd1 titulo: 'Abbey Road'.
cd1 autor:      'The Beatles'.
cd1 origen: 'U.K.'.

cd2 := CD new.
cd2 titulo: 'Tribalistas'.
cd2 autor: 'Tribalistas'.
cd2 origen: 'Brasil'.

cd3 := CD new.
cd3 titulo: 'Peluson of milk'.
```

```
cd3 autor: 'Spinetta'.
cd3 origen: 'Argentina'.

cd4 := CD new.
cd4 titulo: 'Piano Bar'.
cd4 autor: 'Charly García'.
cd4 origen: 'Argentina'.

coleccionCDs add: cd1.
coleccionCDs add: cd2.
coleccionCDs add: cd3.
```

A este conjunto se le pueden enviar algunos mensajes:

```
coleccionCDs size.    "devuelve 3"
coleccionCDs isEmpty.    "devuelve false ya que la colección tiene
                            elementos"
coleccionCDs includes: cd2.   "devuelve true"
coleccionCDs includes: cd4.   "devuelve false"
```

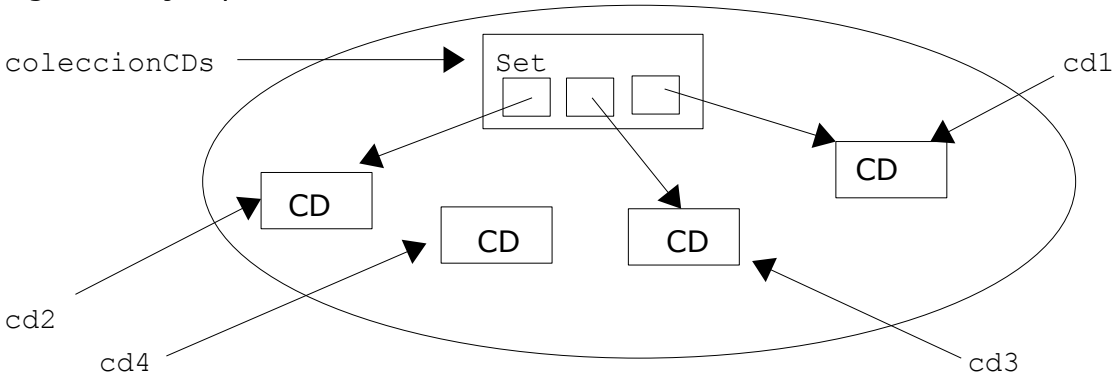
Colecciones y referencias

Es importante aclarar nuevamente que, conceptualmente, la colección no tiene "adentro suyo" a sus elementos. Al agregar un objeto a la coleccción simplemente se agrega una referencia que parte de la colección y llega al objeto "agregado". Es importante destacar también que los objetos que se agrega a una colección pueden estar referenciados por otros objetos que existan y que no necesariamente estén relacionados con la colección.

Yendo más allá del ejemplo que elegimos, los objetos agregados pueden ser incluso otras colecciones, dado que las colecciones son también objetos.

En Smalltalk no estamos limitados en cuanto al tipo de objeto que podemos agregar. Por ejemplo, podemos agregar a nuestra `coleccionCDs` un número, un String, una fecha... lo que sea. Se dice que las colecciones en Smalltalk son heterogéneas dado que permiten que cualquier objeto de cualquier clase pueda ser agregado a ellas junto con otros objetos de otras clases.

Volviendo al ejemplo de la `coleccionCDs`, algunos CDs que creamos son elementos de la colección y además están referenciados por variables. Vemos el siguiente ejemplo clarificador:



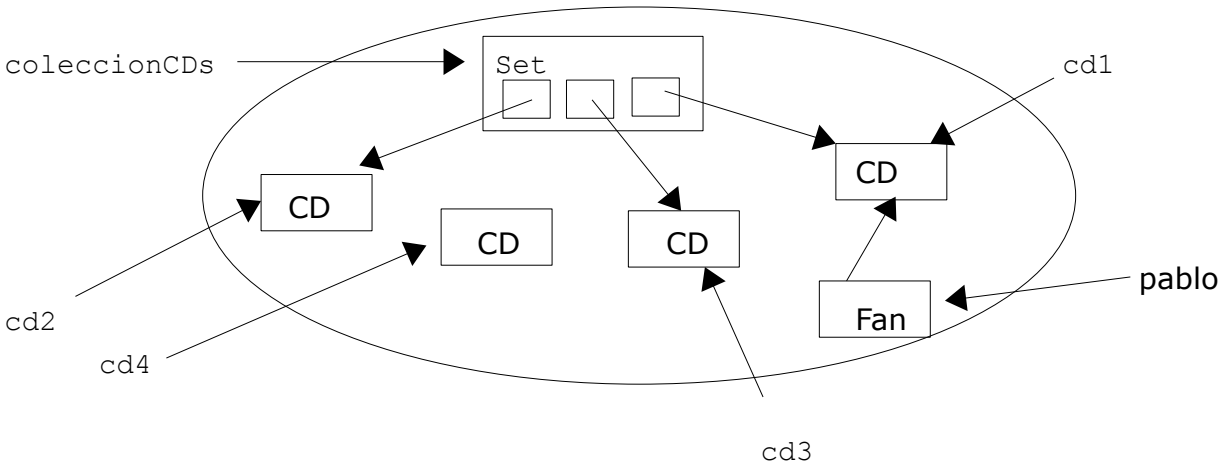
En el gráfico que pusimos arriba, representamos a los objetos con un cuadrado, el "mundo" de los objetos como un círculo, las relaciones de conocimiento como flechas dirigidas, y las variables están por fuera del "mundo" de los objetos.

También se puede ver en el gráfico que el orden en que los CDs se mantienen en el `Set` no corresponde con el orden de inserción de los mismos y tampoco tiene un criterio de orden detectable. Ésta es una de las características que distinguen a un `Set` entre otros tipos de colecciones que existen en Smalltalk: los elementos no están ordenados. La otra característica de un `Set` es que no repite sus elementos, es decir que si agregamos dos veces un mismo objeto, no aparecerá dos veces en él. Otros tipos de colecciones admiten repetidos, mantienen un orden, etc. Los distintos tipos de colecciones los veremos más adelante en este apunte.

Supongamos ahora que está definida la clase Fanatico, que describe un fanático de un CD en particular.

```
pablo := Fanatico new.
pablo cdPreferido: cd1.
```

El gráfico que hicimos antes quedaría de la siguiente manera:



Ahora hay un objeto que tiene tres referencias:

1. es el objeto referenciado por la variable `cd1`
2. es un elemento del `Set`
3. es el CD preferido del fan `pablo`

En el gráfico vemos que 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[2]. Así, los objetos que quedan referenciados por la colección pueden tener otras referencias sin problema. 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).

Es interesante ver que la referencia a un objeto por ser elemento de una

---

2 *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.

colección es suficiente para que el objeto no salga del ambiente cuando pasa el *Garbage Collector*.

Operaciones sobre los elementos

Hay algunas operaciones de colecciones en las que parte de lo que hay que hacer es responsabilidad de cada objeto "dentro" de ella. Por ejemplo, supongamos que quiero obtener, de mi colección de CDs, aquellos que sean de origen nacional. La colección no sabe el origen de cada CD; conoce a los CDs y más aún no sabe qué mensajes puede enviarle. Un `Set` no sabe si lo que tiene son CDs, perros, números, otros `Set`, etc., sólo representa al conjunto, sin saber nada de sus elementos. Por otro lado, cada CD no sabe en qué colección está. Por lo tanto, para resolver el problema de conseguir los CDs de origen nacional, vamos a necesitar a la colección, que sabe qué objetos tiene, y a los CDs, que saben su origen.

Queremos aquellos CDs qque sean elementos de `coleccionCDs`, y cumplan una determinada condición. ¿A qué objeto le vamos a pedir esto? A la colección. El 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 "pedazos de código" son los bloques, en este caso un bloque con un parámetro. Queda así:

```
coleccionCDs select: [:cd | cd origen = 'Argentina' ]
```

El bloque es el objeto que finalmente nos permite realizar lo que queremos, ya que el bloque es el que sabe qué preguntarle a cada CD, y el mensaje select: lo usa para evaluar al bloque y realizar la acción. Cambiemos el código anterior un poco:

```
condicion := [:cd | cd origen = 'Argentina'].
ColeccionCDs select: condicion.
```

Ahora el bloque (que es un objeto) está referenciado por una variable. Entonces puedo evaluar, por ejemplo, la condición para `cd1`.

La `coleccionCDs` 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 `coleccionCDs`). Ésta es la parte que maneja la colección. Los CDs entienden el mensaje `origen`, eso es lo que sabe hacer cada CD.

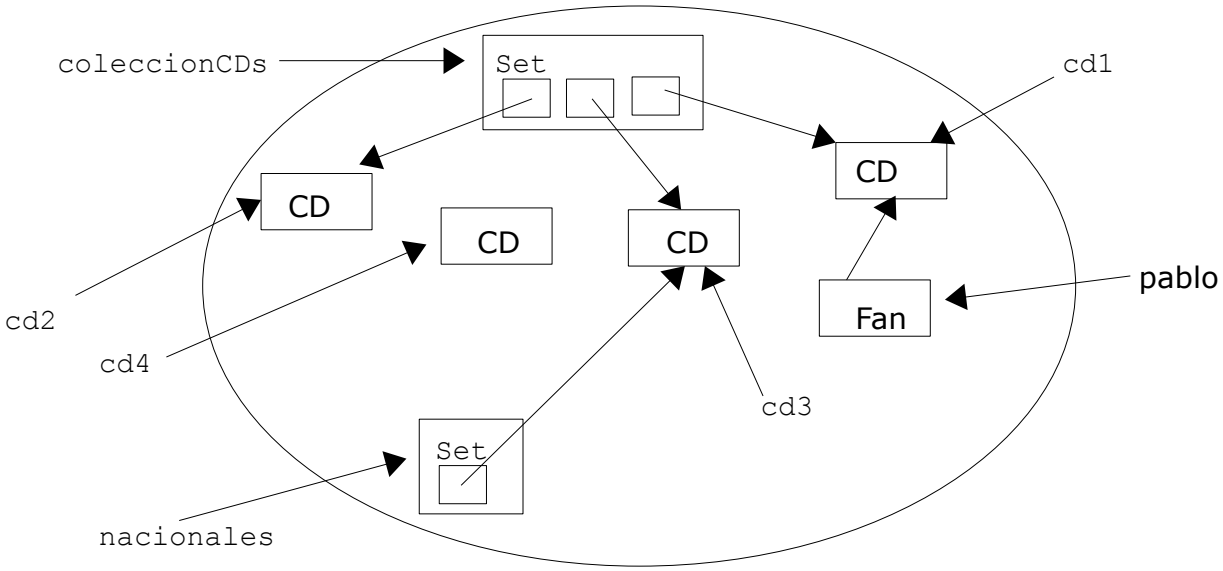
¿Qué devuelve este mensaje? **otra colección**, distinta de `coleccionCDs`, y que tiene como elementos algunos de los objetos que también tiene `coleccionCDs` como elementos; exactamente los que cumplen la condición que le paso como parámetro.

Hemos mencionado que existen distintos tipos de colecciones (que luego veremos), y en este punto nos podemos preguntar: ante el envío del mensaje `select:` ¿qué tipo de colección nos retorna el mensaje? En todos los casos, el tipo de colección va a ser el mismo que el tipo de la colección receptora del mensaje. En particular en este caso, nos va a retornar un nuevo `Set`.

```
nacionales := coleccionCDs select: [:cd | cd origen = 'Argentina']
```

Por lo tanto, podemos asegurar que `nacionales` es un `Set`.

Veamos cómo queda el ambiente luego de evaluar la expresión con el `select:`



En el diagrama anterior, podemos ver:

1. que el resultado del `select:` es una colección distinta de `coleccionCDs`
2. que `coleccionCDs` no se modificó como resultado del `select:`
3. que hay un CD que es elemento de dos colecciones

Y si agregamos a `coleccionCDs` un nuevo CD de origen nacional, ¿se agrega también en la colección referenciada por `nacionales`? La respuesta es NO, porque el conjunto `nacionales` se creó en el `select:` y está separado de `coleccionCDs`. Ambas colecciones son independientes desde el momento que terminó el `select:`.

En resumen: cuando queremos hacer una operación sobre una colección que necesita enviarle mensajes a cada elemento, la operación se la pedimos a la colección, y le vamos a enviar como parámetro un bloque que describe la interacción con cada elemento. Hay varias operaciones de este estilo.

Otros tipos de de colecciones[3]

Smalltalk tiene una clase llamada `Collection` que permite representar colecciones, pero `Collection` es la abstracción superior de la jerarquía de colecciones. De hecho, la clase `Collection` es abstracta y sólo define un protocolo común para todas las colecciones. De ella cuelgan un montón de clases que representan refinamientos de ella (por ejemplo, colecciones que se acceden por índice) o bien colecciones concretas. Al momento de instanciar una colección, tenemos que decidirnos por un tipo particular. En el ejemplo inicial usamos la clase `Set`, que es uno de esos tipos. Veamos el resto:

3 IMPORTANTE*: Leer luego de ver y entender subclasificación y herencia.*

Set y Bag

El primer tipo de colección que visitaremos es el `Bag`. Esta colección es de las denominadas "sin orden" y es la representación de una bolsa. Una bolsa de objetos. El `Bag` es la colección más adecuada para representar por ejemplo un carrito o bolsa de supermercado. Imagínense que puedo poner tres latas de tomates, dos botellas de agua y siete peras.

El segundo tipo que veremos es el `Set` (sí, el que está más arriba en el ejemplo). El `Set` está concebido para la representación de conjuntos. Una propiedad distintiva sobre conjuntos es que los elementos que pertenecen al conjunto son únicos en él. Es decir, en los conjuntos no vamos a tener dos veces el mismo elemento (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 vamos a representar así:

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

Bien, tenemos el objeto unaLata y queremos 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 representamos a mi carrito de compras con un `Bag`, (o sea que hice anteriormente `miCarrito:= Bag new.`) entonces sí voy a tener 3 objetos dentro de mi carrito.

En cambio, si representáramos mi carrito de compras con un `Set` (o sea que hice anterioremente `miCarrito:= Set new.`) entonces hubiese tenido un solo objeto dentro de mi colección. Simplemente, cuando hacemos el segundo `miCarrito add: unaLata` el `Set` identifica que ya tiene ese objeto en la colección y no lo vuelve a agregar. Ésto que acabamos de mencionar se puede comprobar fácilmente enviando el mensaje `size` a miCarrito y viendo que en el caso de un `Bag` el tamaño es 3 y en el

caso de un `Set` el tamaño es 1.

Colecciones Ordenadas

Antes que nada, debemos preguntarnos ¿Qué quiere decir que una colección esté "ordenada"? Básicamente que hay un elemento 1, un elemento 2, etc., al contrario de un `Set` o un `Bag`, en donde los elementos están "todos tirados". Es decir que podemos acceder a los elementos de cualquier colección ordenada (veremos que hay varias variantes) parecido a como se accede a un array (arreglo) o una lista.

Para acceder a un objeto particular en una posición se le envía a la colección el mensaje `at:`, con un entero como parámetro, que indica la posición a la que quiero acceder. Entonces si queremos el cuarto elemento de `miCol` que es una colección ordenada, lo podemos pedir así:

```
miCol at: 4
```

Además las colecciones ordenadas tienen definidos los mensajes `first` y `last` que devuelven el primero y el último elemento.

Una de las variantes más usadas de colecciones ordenadas es la `OrderedCollection`. Esta colección ordena sus elementos, y el criterio es el orden en cual fueron agregados a la colección.

Si creamos mi colección así:

```
miCol := OrderedCollection new.
miCol add: 'esto'.
miCol add: 'es'.
miCol add: 'un'.
miCol add: 'ejemplo'.
```

Después podemos pedirle varias cosas:

`miCol first`	el primero	`'esto'`
`miCol last`	el último	`'ejemplo'`
`miCol at: 3`	el tercero	`'un'`
`miCol at: 4`	el cuarto	`'ejemplo'`
`miCol size`	cantidad de elementos	`4`

Y podemos seguir agregándole elementos, ante lo cual la colección "se estira", pues tiene un tamaño variable (a diferencia de un array).

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

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

Existe también un tipo de colección llamada `SortedCollection`, que a diferencia de la primera, el criterio de ordenamiento puede ser definido. Si no

definimos el criterio, la `SortedCollection` ordena los elementos por su "orden natural" (significa que los ordenará de menor a mayor, usando el mensaje <). 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 ya creada. La forma de hacerlo, es pasarle lo que denominamos "`sortBlock`", que es un bloque (objeto de la clase `BlockClosure`). También podemos crear una colección con el bloque ya definido, mandándole el mensaje `sortBlock:` a la clase.

Arreglos

Finalmente, nos queda ver el conocido `Array`. En Smalltalk también existe, y una de sus características distintivas es que es de tamaño fijo. Para instanciar un `Array`, hacemos `Array new: 6`, donde `6` es la cantidad de elementos que contendrá. En casos especiales donde los elementos son literales booleanos, strings o numeros, se pueden crear los arreglos de la siguiente forma:

```
arregloDeLiterales := #(1 2 'a' true).
```

Los Arrays no implementan el mensaje `add:`, justamente porque no se pueden 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.
```

De un tipo a otro

Todas las colecciones entienden una serie de mensajes que permiten obtener distintos tipos de colecciones con los mismos elementos que la colección receptora. 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`. Hacemos lo siguiente:

```
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`.

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", dado por el mensaje <. En este caso, debemos tener en cuenta que todos los objetos que agreguemos a esta colección deberán entender el mensaje <, pues si así no fuera, tendríamos un error de mensaje no entendido.

Acerca del sort block

El `sortBlock` es un bloque que necesita 2 parámetros, que son los objetos a comparar. 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.

Diccionarios

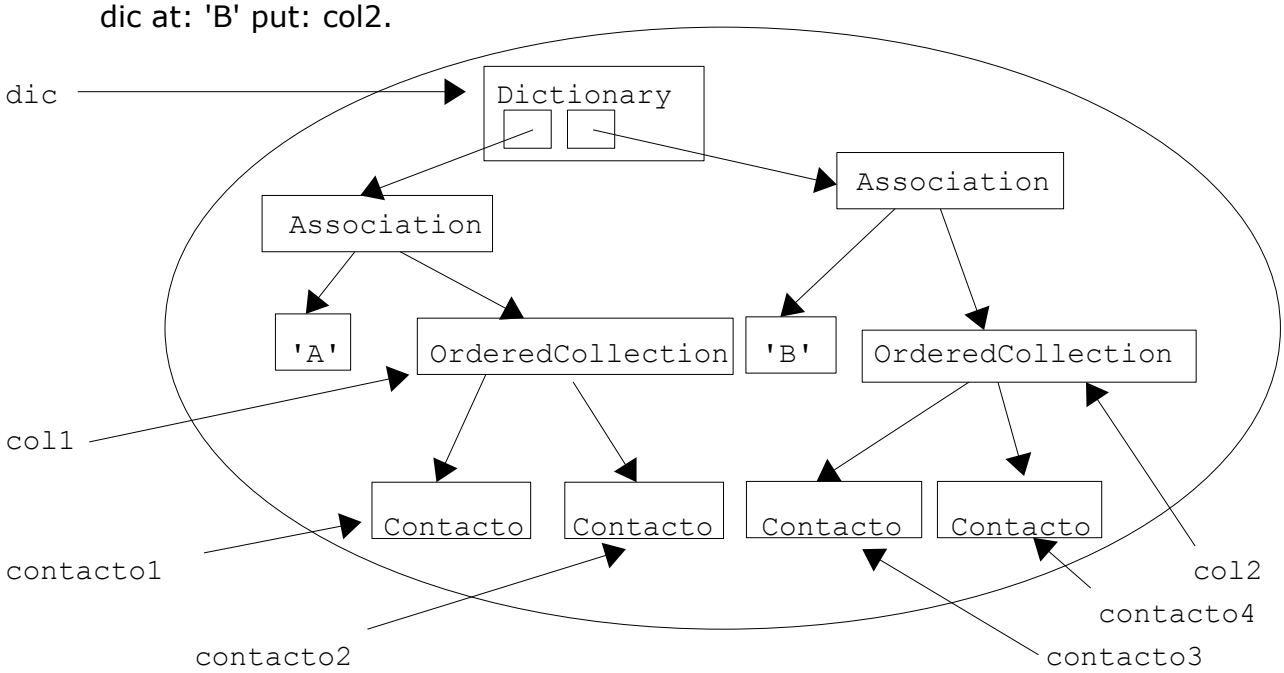
Los diccionarios (`Dictionary`) son un tipo especial de colecciones, ya que sus elementos son pares de claves y valores. Los objetos que referencia esta colección son instancias de una clase particular: `Association`. En una `Association` se tienen dos objetos, uno que juega el rol de clave y el otro el valor. En particular el objeto `nil` no puede ser la clave de una asociación de un diccionario.

Con los diccionarios se pueden representar obviamente diccionarios de palabras o sinónimos, pero también se pueden representar cosas interesantes como una agenda, donde cada elemento del diccionario es una letra, y el valor asociado son todos los contactos que empiezan con esa letra.

Otra forma de ver a un diccionario es como una lista, donde el acceso a los elementos no se hace por índice entero sino por cualquier otro objeto. De hecho, el mensaje para agregar nuevos elementos al diccionario es `at: put:`, pero en este caso el primer parámetro puede ser cualquier objeto.

Veámoslo gráficamente con un ejemplo de código:

```
dic := Dictionary new.
col1 := OrderedCollection new.
contacto1 := Contacto new: 'Armendariz Garcia' nombre: 'Joaquín'
                nro:'8122967'.
contacto2 := Contacto new: 'Andrade Cantu' nombre: 'Rolando' nro:'913-3122'.
col1 add: contacto1.
col1 add: contacto2.
dic at: 'A' put: col1.
col2 := OrderedCollection new.
contacto3 := Contacto new: 'Blanco Castillo' nombre: 'Juan' nro:''.
contacto4 := Contacto new: 'Barrera Bravo' nombre: 'Felipe' nro:'8120298'.
col2 add: contacto3.
col2 add: contacto4.
dic at: 'B' put: col2.
```



Observar que el diccionario crea una instancia de Association y que cada instancia referencia a la clave y valor que se agregaron al diccionario.

A diferencia del resto de las colecciones, los diccionarios definen métodos que permiten realizar operaciones sobre las asociaciones, sobre los objetos que son clave y sobre los objetos que son valor. Se invita al lector a averiguar en la jerarquía de Collection de Smalltalk cuáles son los mensajes que se define en los diccionarios para tales fines.

Resumen de mensajes que entiende una colección

¿Qué puedo hacer con los conjuntos de objetos?

Agregarle un objeto

¿Qué mensaje envío? `add:`

Un ejemplo: `pajaros add: pepita.`

Otro ejemplo: `usuarios add: usuario23.`

Agregarle varios objetos de una sola vez

¿Qué mensaje envío? `addAll:`
Un ejemplo: numerosPrimos `addAll: #(2 3 5 7 11).`
Otro ejemplo: `usuarios addAll: (OrderedCollection with: usuario23).`

Quitarle un objeto

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

Quitarle varios objetos de una sola vez

¿Qué mensaje envío? `removeAll:`
Un ejemplo: `pajaros removeAll: (OrderedCollection with: pepita with: pepito).`
Otro ejemplo: `numerosPrimos removeAll: #(1 -1 0).`

Preguntarle si tiene un objeto

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

Preguntarle la cantidad de veces que tiene un objeto

¿Qué mensaje envío? occurencesOf:
¿Qué me devuelve? Un objeto entero
Un ejemplo: `pajaros occurencesOf: pepita.`
Otro ejemplo: `usuarios occurencesOf: usuario23.`

Preguntarle si está vacía

¿Qué mensaje envío? isEmpty
¿Qué me devuelve? Un objeto booleano, `true` o `false`
Un ejemplo: `pajaros isEmpty.`

Preguntarle si tiene al menos un objeto

¿Qué mensaje envío? notEmpty
¿Qué me devuelve? Un objeto booleano, `true` o `false`
Un ejemplo: `pajaros notEmpty.`

Recorrer los elementos, realizando alguna acción que cada uno de ellos

¿Qué mensaje envío? `do:`
¿Qué me devuelve? La colección receptora.
Un ejemplo: `pajaros do: [:unPajaro | unPajaro cantar ].`
Otro ejemplo: `usuarios do: [:unUsuario | unUsuario depositar: 100 en: miCuenta].`

Seleccionar los elementos que cumplen con un criterio

¿Qué mensaje envío? `select:`
¿Qué 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

¿Qué mensaje envío? `reject:`
¿Qué 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

¿Qué mensaje envío? `collect:`
¿Qué 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

¿Qué mensaje envío? `allSatisfy:`
¿Qué 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

¿Qué mensaje envío? `anySatisfy:`
¿Qué 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].
```

Obtener un objeto de la colección que cumple con un criterio

¿Qué mensaje envío? detect: `ifNone:`
¿Qué me devuelve? un objeto si alguno de los objetos de la colección da `true` al evaluarlo con el bloque, o evalúa el bloque que se leenvía como segundo parámetro en caso contrario y retorna lo que retorna el bloque.
Un ejemplo: `pajaros detect: [:unPajaro | unPajaro estaDebil] ifNone:['No hay pájaros débiles'].`
Otro ejemplo:

```
numerosPrimos detect: [:nro | nro even] ifNone:['El 2 debería estar definido como primo!']
```

Evaluar un bloque binario sobre la colección, usando como primer parámetro la evaluación previa, y como segundo parámetro el elemento actual. El resultado se usará como primer parámetro en la siguiente evaluación.

¿Qué mensaje envío? `inject:into:`
¿Qué 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]]
```

Cómo elegir la colección más adecuada?

A continuación se muestra un algoritmo posible a seguir para seleccionar la colección más adecuada de acuerdo al problema a modelar. Sin embargo, la elección de la colección debería hacerse de manera intuitiva, considerando las características de cada una de las opciones disponibles.