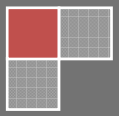
2013

Colecciones con JAVA

Laboratorio II - Programación II

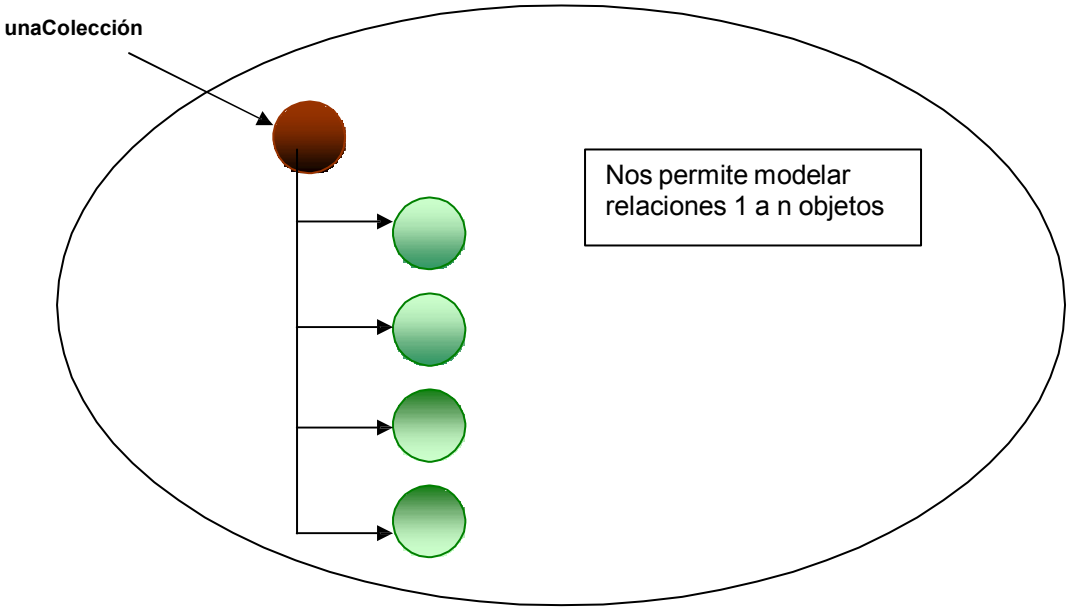
Justo Miguel Vargas
Universidad Tecnológica Nacional - Facultad Regional Resistencia
09/10/2013



Indice

Definición

Una colección es el objeto que representa un conjunto de elementos relacionados.



¿Qué le puedo pedir a una Collection?

. agregar un elemento
. quitarlo
. saber si un elemento está en una colección
. su tamaño

Un ejemplo en código generando una colección de escritores:

```
Escritor bioy = new Escritor("Adolfo Bioy Casares");

Collection escritores = ... algo ...
escritores.add(bioy);
escritores.add(new Escritor("Julio Cortazar"));
```

Presentando Generics

Revisando el contrato de Collection, ¿qué parámetro espera la colección cuando hago add? Y… en principio le paso un escritor, pero podemos tener una colección de facturas, de exámenes, etc. Entonces tengo dos opciones:

1. Digo que el add recibe un Object (dejo el contrato para que acepte un elemento lo más general posible)
2. Creo una clase EscritoresCollection cuyo contrato es `public void add(Escritor escritor)` (y lo mismo para cada colección diferente de empleados, de clientes, etc.)

Mmm… ¿qué tal si se pudiera parametrizar un tipo en la definición de una clase? Es decir, si en lugar de tener:

```
Collection
public void add(Object o)
```

pudiéramos definirlo como:

```
Collection<E>
public void add(E element)
```

donde E es un tipo (clase, interfaz) que restringe el contrato de add, de manera que:

```
escritores.add(new Banana());  // una Banana no tiene relación con Escritor
```

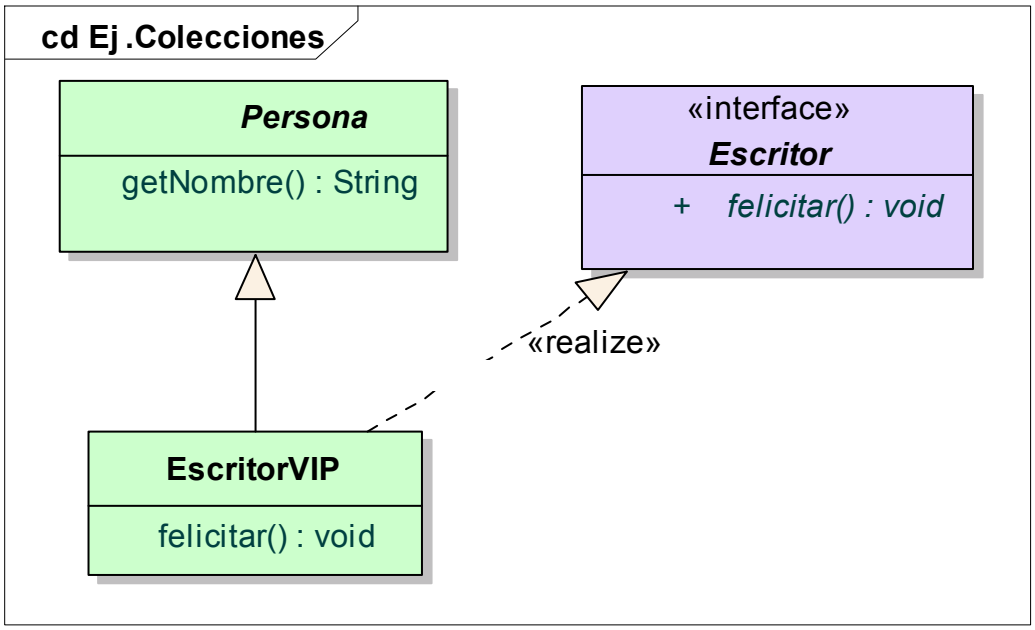
no compile.

Volviendo sobre esta definición, el código de la página anterior debería cambiar: ya no estamos diciendo que escritores es "una colección", ahora queremos decir que escritores es una colección de escritores:

```
Collection<Escritor> escritores = ... algo ...
```

Es decir, estamos agregando una restricción: todos los elementos deben poder castearse al tipo Escritor.

Nuevamente insistimos, Escritor es un tipo, no necesariamente una clase. Si tuviéramos una clase EscritorVIP que implementa la interfaz Escritor



podríamos hacer algo como:

```
Collection<Escritor> medidas = ... algo
... escritores.add(new EscritorVIP("Julio
Cortazar"));
```

Esta posibilidad (recibir tipos como parámetros en la definición de una clase o interfaz) nace a partir de la JDK 1.5 y se llama **Generics**[1].

Si bien podemos parametrizar el tipo para definir colecciones de alumnos, clientes, etc. todos los elementos tienen que cumplir algo fundamental: ser objetos. La primera consecuencia molesta para el desarrollador es que los tipos primitivos no se pueden agregar a una colección[2], entonces es necesario utilizar clases **Wrappers** que "envuelvan" los tipos primitivos transformándolos en objetos. Así tenemos al Integer en lugar del int, al Float en lugar del float, etc. La operación que convierte un tipo primitivo a un objeto también se conoce como **Box**.

Ejemplo:

```
Collection<Number> medidas = ... algo ...
Integer edad = new Integer(21);          □ "wrappeo" un entero para poder agregarlo
medidas.add(edad);
```

Resulta algo tedioso tener que hacer un *box* de los tipos primitivos cada vez que tenemos que agregarlo a la colección y un *unbox* cuando lo recuperamos para trabajarlo como tipo primitivo nuevamente.

Autoboxing

A partir de las versiones 1.5 de Java ya no necesitamos hacer la conversión nosotros sino que el compilador se encarga de hacerlo automáticamente (por eso lo de autoboxing):

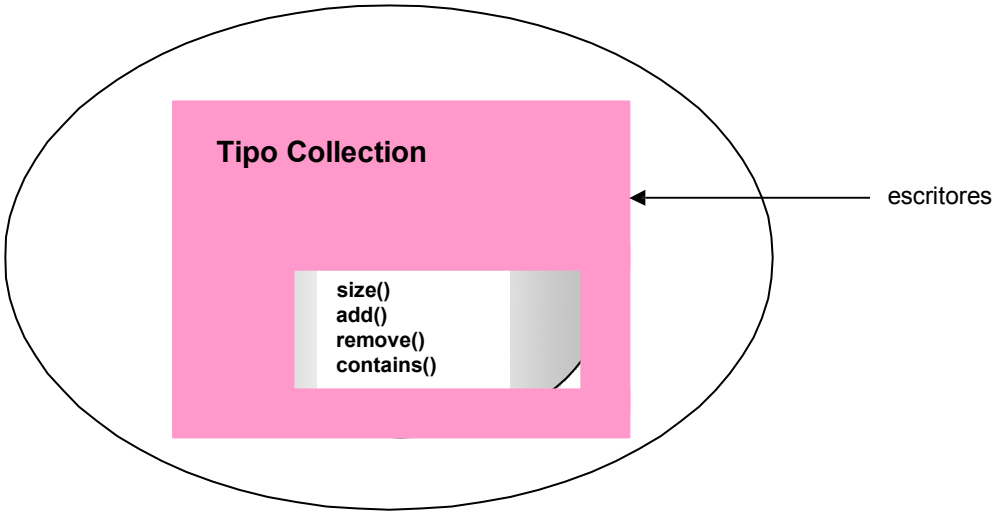
```
// Box automático
Collection<Integer> medidas = ... algo ...
medidas.add(21);                         □ el compilador convierte de int a Integer
                                           automágicamente

// Unbox automático
Collection<Integer> medidas = ... algo ...
int pepe = <obtengo un elemento>;        □ el compilador convierte de Integer a int
                                           automáticamente
```

No obstante debemos tener precaución dado que un Integer no es lo mismo que un int: un Integer puede referenciar a `null` mientras que un int vale 0 si no está inicializado[3]

[1] Para mayor referencia, puede verse el tutorial http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
[2] Recordamos algunos tipos primitivos: byte, short, int, long, float, boolean, char
[3] Para mayor referencia consultar http://java.sun.com/j2se/1.5.0/docs/guide/language/autoboxing.html

Contrato de Collection

Collection es una interfaz, no la implementación final de la clase. Yo puedo definir a la variable escritores de "tipo" Collection, más allá de la clase real a la que pertenezca:



¿Por qué no aparecen los métodos para recuperar al cuarto o al quinto elemento, o para agregar un objeto en el segundo lugar? Porque tendría que conocer cómo está guardado internamente cada elemento, o asegurarme que en el mismo orden que ingreso los elementos los recupero. **Corolario:**

usar Collection no me garantiza el orden de los elementos

Ok, la mala noticia es que tengo menos información del objeto. Pero ¿cuál es la ventaja? Logro un menor acoplamiento entre el cliente que usa la colección y el que la implementa.

Colecciones en Java

Hacer lo mismo en Java es casi tan feliz como en lenguajes puros OO:

```
Collection<Escritor> escritores = ...

...

for (Escritor escritor : escritores) {
    escritor.felicitar();
}
```

Iteradores internos y externos

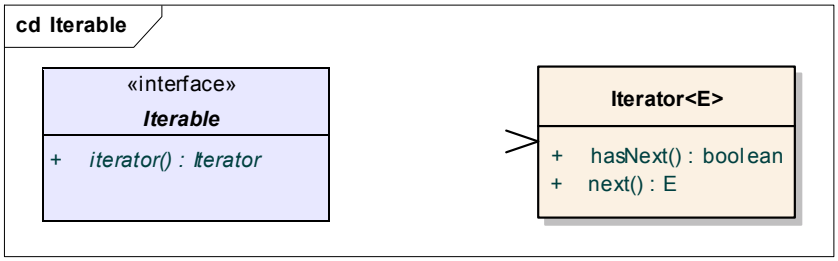
En el ejemplo de la página anterior el loop for-each nos permite recorrer los elementos de esa colección, entonces decimos que el Iterador es **interno**. Para recorrer internamente los elementos el for requiere que la colección se pueda recorrer

```
for (Escritor escritor : escritores) {
```

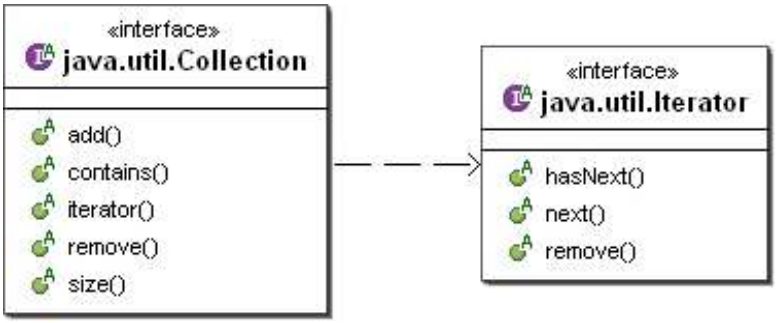
debe implementar la interfaz Iterable

```
public interface Collection extends Iterable {
        …
}
```

La interfaz Iterable define que debe implementar un método iterator()



¿Y qué representa un objeto Iterator? Bueno, es un objeto que sabe cómo se recorren los elementos de una colección. De hecho se puede separar el almacenamiento de los elementos y la forma en que se recorren, y trabajar así con un Iterador **externo**:



Como acabamos de ver, a una colección se le puede pedir SU Iterator (porque el contrato de Collection dice que implementa Iterable):

¿Qué entiende un Iterator<E>?

- hasNext() : devuelve un boolean (si hay más elementos)
- next() : devuelve un E (está definido en función al tipo de la Collection)

Dos formas de resolver el mismo ejemplo de los escritores:

Con un for:

```
for (Iterator<Escritor> it = escritores.iterator(); it.hasNext(); ) {
Escritor escritor = it.next();
     escritor.felicitar();
}
```

Con un while:

```
Iterator<Escritor> it = escritores.iterator();
while (it.hasNext()) {
    Escritor escritor = it.next();
    escritor.felicitar();
}
```

Vemos que el contrato del iterador it define que next() devuelve un Escritor:

```
while (it.hasNext()) {
    Escritor escritor = it.next();
    escritor.felicitar();
```

Escritor java.util.Iterator.next()

¿Qué diferencias encontramos?

- Los iteradores internos son más fáciles de usar (sobre todo en tareas repetitivas y recurrentes como calcular totales de facturación de un cliente, filtrar facturas impagas de un proveedor y mostrar las materias a las que se inscribió un alumno)
- Al trabajar con iteradores externos cada vez que yo hago next(), estoy avanzando al siguiente elemento del iterador. La primera consecuencia fácil de advertir es que hay efecto colateral. La segunda es que tengo que tener cuidado de no hacer un next() equivocadamente en el contexto de una iteración.

```
Iterator<Escritor> it = escritores.iterator();
while (it.hasNext()) {
    if (it.next().esBueno()) {
        it.next().felicitar();  ¡CHAN!
    }
}
```

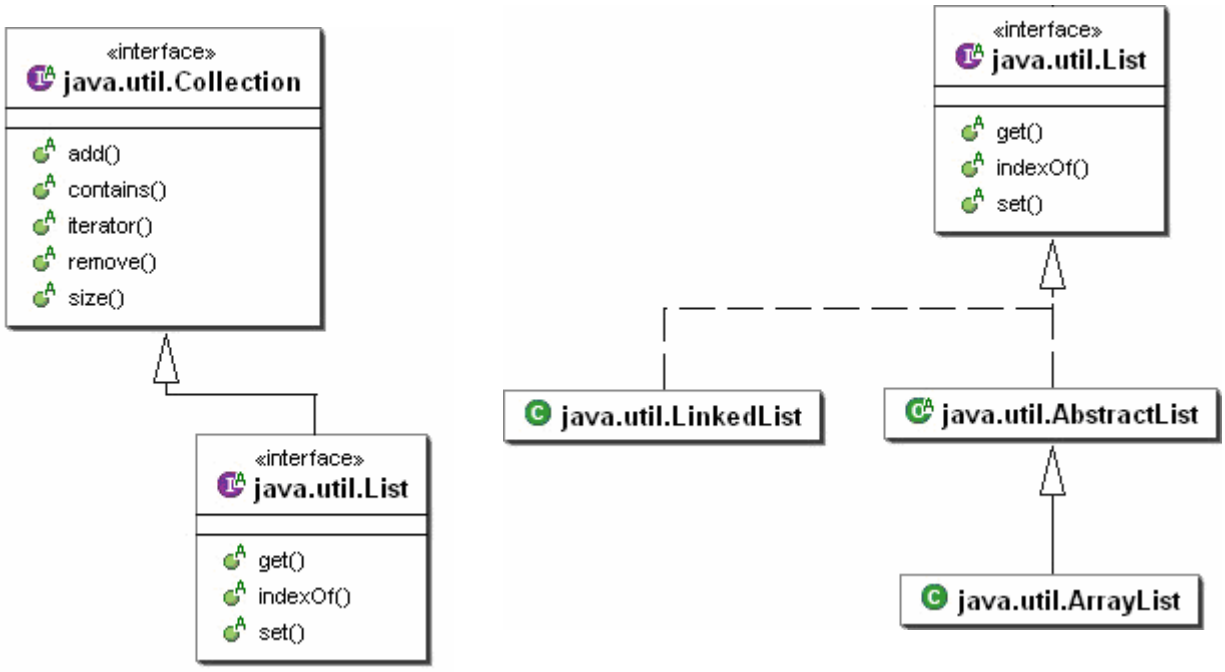
- Separar al objeto que almacena los elementos y al que los accede supone una ventaja si la lógica para recorrer los elementos no es trivial (tengo mayor control sobre el algoritmo que trae cada elemento)[4].

```
Collection<Escritor> escritores = ...
```

¿Qué va del lado derecho? ¿Qué tipo de colección me conviene elegir? Y bueno hay para todos los gustos.

Implementaciones de Collection

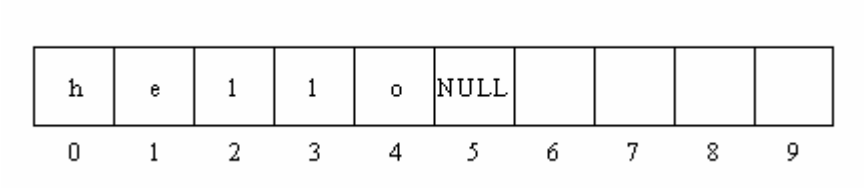
Interface List




```
Collection<Escritor> c = new ArrayList<Escritor>();
```

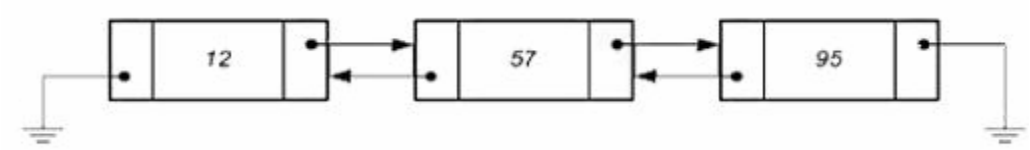
- Son colecciones de objetos indexadas por un número que representa la posición de cada objeto dentro de la misma.
- A diferencia del contrato de Collection, donde no se asegura el orden en que se almacenan o recorren los elementos, el contrato de List me asegura que el orden en que se ingresan será el mismo orden en que se lean los elementos. Este contrato es débil, si una implementación de List rompiese este contrato el compilador no se daría cuenta pero el funcionamiento del programa puede no ser el esperado.
- Al recorrerlas con el iterator común, o por posición/índice las mismas se verán recorridas (inicialmente) en el orden de agregado de los elementos.
- Una List, ¿entiende contains()? Sí, porque es una Collection.

Clase Array List



- Almacena la información internamente en un vector.
- Es ideal para realizar acceso aleatorio a los elementos de la colección.
- Para guardar un nuevo elemento, si no tenemos espacio disponible en el vector interno, el mismo será redimensionado (aumentado su tamaño total en un 50% o 100% según la implementación), este proceso es pesado por lo cual no es recomendable utilizar esta implementación si nos dedicaremos principalmente a agregar y borrar elementos.
- Para agregar o borrar un elemento en una posición que no sea la última de la lista, se moverán los elementos que estén a la derecha, lo cual en un vector grande también es un proceso pesado.
- Podemos definir en el constructor del ArrayList el tamaño inicial del vector interno (si sabemos aproximadamente la cantidad de elementos que almacenaremos en el mismo), esto lógicamente mejora la performance.

Clase LinkedList



- Almacena la información internamente en una lista doblemente enlazada.
- Es ideal si se desean realizar múltiples agregados y borrados de elementos, dado que cada inserción o eliminación realiza una cantidad mínima de cambios, a diferencia de ArrayList que tiene que mover todos los elementos siguientes por el vector.
- El acceso secuencial es un poco menos performante que en una ArrayList.
- Para acceder a un elemento de la lista por índice se recorrerán todos los elementos que estén antes del mismo, por lo cual el rendimiento del acceso por índice es muy inferior al del ArrayList.

Referencias y casteos

```
// Así no funciona ®
Collection<Escritor> c = new ArrayList<Escritor>();
List<Escritor> l = c;
```

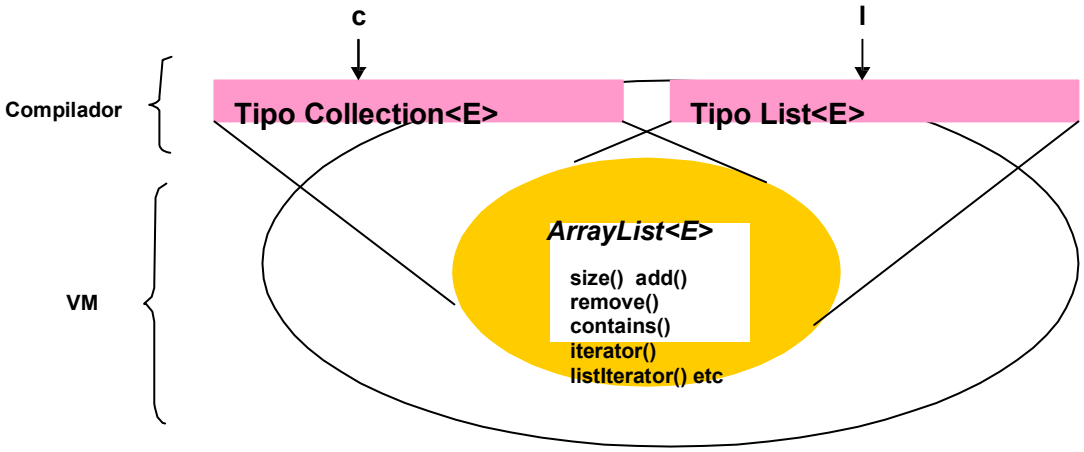
No puedo hacer que la variable l vea más información que la original, sin hacer downcast:

```
List<Escritor> l = (List<Escritor>) c;
```

En cambio el compilador no tiene objeciones para pasar de una variable de mayor visibilidad (la interfaz List) a otra de menor (la interfaz Collection).

```
// Así sí funciona ☺
List<Escritor> l = new ArrayList<Escritor>();
Collection<Escritor> c = l;
```

Vemos el ejemplo en el ambiente, donde el objeto referenciado **nunca deja de ser un ArrayList**, sólo que me interesa verlo de diferentes maneras:



Está claro que el compilador no me dejará enviarle este mensaje
`c.listIterator();`
ya que c es de tipo Collection y Collection no tiene declarado ese mensaje (aún cuando el ArrayList lo tiene definido en su comportamiento).

Por otra parte ambas variables referencian **al mismo objeto**, si el ArrayList está vacío y yo hago

```
c.add(new Escritor("Julio Cortazar"));
```

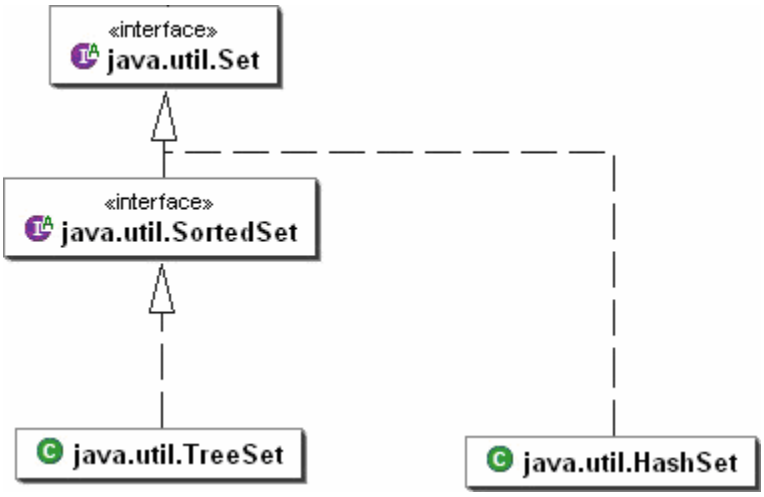
Cuando le pregunto

```
System.out.println(l.size());
```

por supuesto devuelve 1.

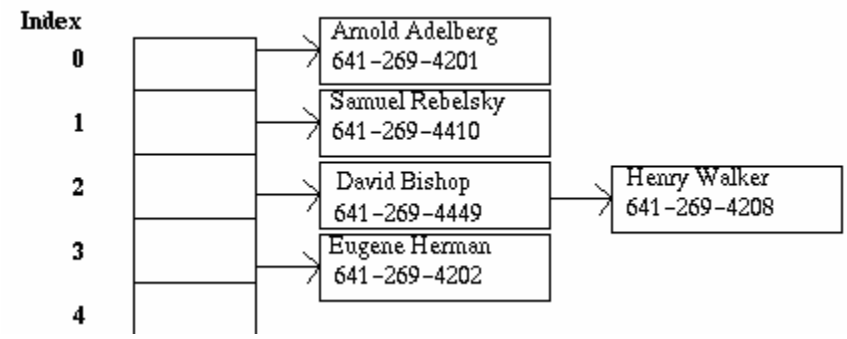
No es que se actualizan ambos objetos: ***no hay dos objetos, sino que hay dos variables referenciando al mismo objeto***.

Interface Set



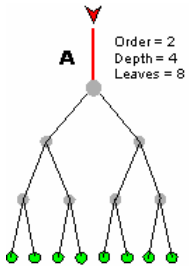
- Son colecciones de objetos donde no puede haber dos elementos iguales.
- La interfaz es una interfaz vacía, no agrega ningún método a Collection. La diferencia está en que el contrato de Set dice que los elementos no deberán repetirse.
- El contrato es débil, puede romperse si implementamos un Set propio que permita ingresar elementos duplicados (eso no está bueno que suceda, deberíamos respetar el contrato de las interfaces que implementamos o no suscribir al contrato de Set).

Clase HashSet



- Implementa el conjunto de datos utilizando una tabla hash.
- Utiliza en consecuencia funciones de hashing para el posicionamiento y búsqueda de los elementos.

Clase TreeSet



- Implementa la interfaz SortedSet, que agrega como parte del contrato que los elementos del Set se almacenarán y recorrerán en un orden especificado.
- El ordenamiento de los elementos se va dando automáticamente a medida que son ingresados.
- Almacena la información internamente en un árbol.
- El orden de los elementos puede estar dado por el orden natural, para lo cual los elementos deben implementar *Comparable*, o por un orden específico pasándole como parámetro al constructor del TreeSet un Comparator (ver Comparación/Ordenamiento de una Colección en el presente apunte)

Si ven el Javadoc de java.util.Set, van a ver que dice algo de 'a.equals(b)'.

equals

public boolean equals(Object obj)

Determine whether this Object is semantically equal to another Object.
There are some fairly strict requirements on this method which subclasses must follow:

- *It must be transitive. If* `a.equals(b)` *and* `b.equals(c)`*, then* `a.equals(c)` *must be true as well.*
- *It must be symmetric.* `a.equals(b)` *and* `b.equals(a)` *must have the same value.*
- *It must be reflexive.* `a.equals(a)` *must always be true.*
- *It must be consistent. Whichever value a.equals(b) returns on the first invocation must be the value returned on all later invocations.*
- `a.equals(null)` *must be false.*
- *It must be consistent with hashCode(). That is,* `a.equals(b)` *must imply* `a.hashCode() == b.hashCode()`*. The reverse is not true; two objects that are not equal may have the same hashcode, but that has the potential to harm hashing performance.*

This is typically overridden to throw a ClassCastException if the argument is not comparable to the class performing the comparison, but that is not a requirement. It is legal for `a.equals(b)` *to be true even though* `a.getClass() != b.getClass()`*. Also, it is typical to never cause a NullPointerException.*
In general, the Collections API (`java.util`*) use the* `equals` *method rather than the* `==` *operator to compare objects. However, java.util.IdentityHashMap is an exception to this rule, for its own good reasons.*
The default implementation returns `this == o`*.*

---

hashCode

`public int` **`hashCode`**`()`

Get a value that represents this Object, as uniquely as possible within the confines of an int.

There are some requirements on this method which subclasses must follow:

- Semantic equality implies identical hashcodes. In other words, if `a.equals(b)` is true, then `a.hashCode() == b.hashCode()` must be as well. However, the reverse is not necessarily true, and two objects may have the same hashcode without being equal.
- It must be consistent. Whichever value o.hashCode() returns on the first invocation must be the value returned on all later invocations as long as the object exists. Notice, however, that the result of hashCode may change between separate executions of a Virtual Machine, because it is not invoked on the same object.

Notice that since `hashCode` is used in java.util.Hashtable and other hashing classes, a poor implementation will degrade the performance of hashing (so don't blindly implement it as returning a constant!). Also, if calculating the hash is time-consuming, a class may consider caching the results.

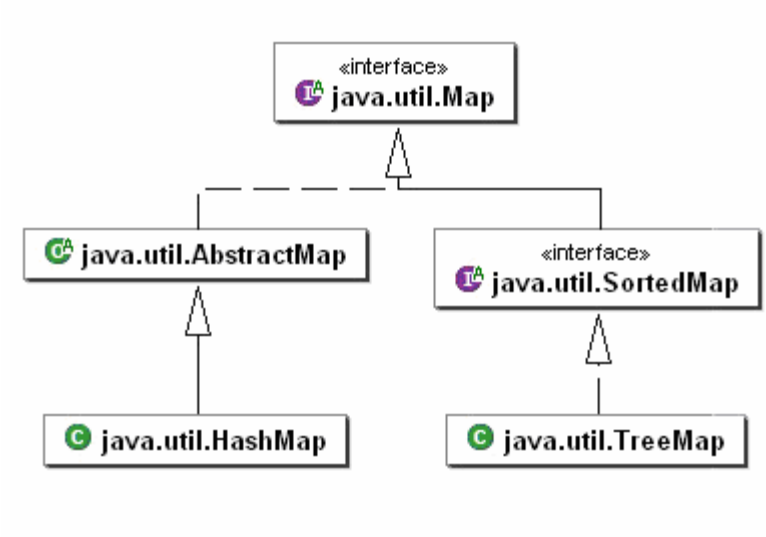
The default implementation returns `System.identityHashCode(this)`

Eso es porque en algunos casos, podría haber dos o más objetos que representan al mismo ente (ver definición de objeto de PDP). ¿Esto está bien, está mal, conviene, no conviene? Eso lo vamos a hablar en otro apunte[5]. En lo que sigue de éste, vamos a entender eso como 'a y b son el mismo objeto' (*léase*: no vamos a redefinir el equals y por ende no vamos a necesitar redefinir el hashCode, si tengo dos objetos Persona es porque representan distintos entes, por más que se llamen igual).

Interface Map

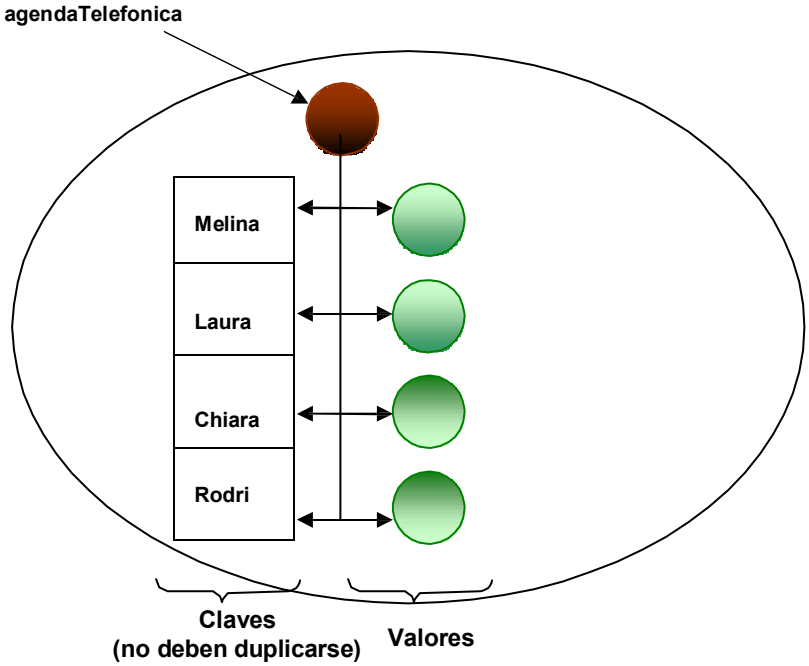
El concepto es equivalente al Dictionary de Smalltalk, me permite trabajar con un par clave/valor. Una consecuencia de esto es que Map no cumple el contrato de Collection, ya que para agregar elementos necesita dos valores (no le sirve implementar el add de Collection). Por eso existe el mensaje put(K clave, V valor), donde K y V son los dos tipos que necesito pasarle al Map cuando lo defino:

```
Map<String, Persona> agendaTelefonica = new HashMap<String, Persona>();
agendaTelefonica.put("Pocha", new Persona("Scarlett",
                                          "Johansson", "4550-1291"));
agendaTelefonica.get("Pocha")    // Se accede por clave
```



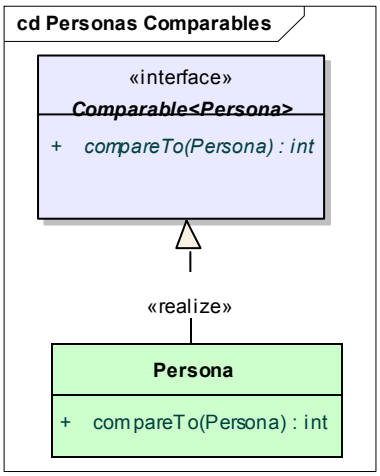
Se puede obtener el Set de claves (`keySet()`), una Collection de valores (`values()`) o un Set de Entries (`entrySet()`, un Entry es un par Clave-Valor).

Un Mapa en el ambiente de objetos



Comparación / Ordenamiento de una colección

La interfaz Comparable<T> nos permite relacionar objetos para poder darles un ordenamiento natural. Si por ejemplo decimos que las personas implementan la interfaz Comparable<T> donde T será el tipo Persona:



esto significa que debemos implementar el método compareTo() en Persona. El contrato de este método nos dice que debe devolver:

- un número negativo si el objeto receptor es menor que el objeto recibido como argumento
- un número positivo si el objeto receptor es mayor que el objeto recibido como argumento, o
- cero si ambos son iguales.

Una implementación posible: en la definición de la clase tenemos que decir que Persona implementa la comparación de personas:

```
public class Persona implements Comparable<Persona> {
```

Y definir el método compareTo():

```
public int compareTo(Persona persona) {
     if (this.getEdad() == persona.getEdad()) {
          return 0;
     }
     if (this.getEdad() > persona.getEdad()) {
          return 1;
     } else {
          return -1;
     }
}
```

Otra implementación posible (`getEdad()` nos devuelve un objeto Integer, no un `int`):

```
public int compareTo(Persona persona) {
     return this.getEdad().compareTo(persona.getEdad());
}
```

Algunas observaciones:

- Elegir números positivos y negativos no es una decisión feliz como criterio para comparar objetos. De todas maneras en el contrato está claramente documentado.
- La interfaz Comparable me sirve para comparar personas sólo por edad: es el tipo de ordenamiento natural que debería usar por defecto. Ahora, si yo quiero ordenar alfabéticamente una lista de personas, ¿qué otra opción tengo?

+ Comparable<T> -+ una sola forma de ordenamiento.

+ Comparator<T> -+ me permite definir más formas de ordenamiento.

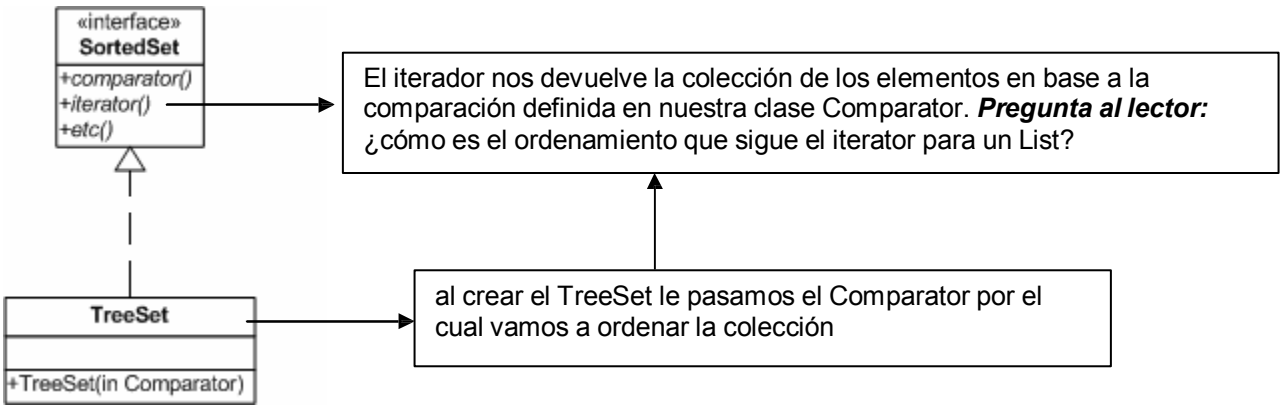
```
...

public class ComparadorDePersonasPorEdad implements Comparator<Persona> {
     public int compare(Persona persona1, Persona persona2) {
           return persona1.getEdad().compareTo(persona2.getEdad());
     }
}
```

```
...

public class ComparadorPersonasPorNombre implements Comparator<Persona> {
     public int compare(Persona persona1, Persona persona2) {
           return persona1.getNombre().compareTo(persona2.getNombre());
     }
}
```

¿Y cómo lo usamos?

```
Collections.sort(personas);          □ ver java.util.Collections, utiliza por
                                     default elementos comparables

Collections.sort(personas, new ComparadorPersonasPorNombre());
                                     □ también puedo usar comparators (o
                                        crear y usar comparators anónimos)
```

- SortedSet / SortedMap , a quienes les pasamos de qué manera queremos ordenar las personas que queremos. Ejemplo:



Más material para investigar

- **http://java**.sun.com/j2se/**1.5**.0/docs/api/**java**/util/**Collection**s.html
- Commons-Collections, CollectionUtils (**jakarta**.**apache**.org/).
- Sincronización de colecciones: manejo de concurrencia/inmutabilidad